



AMD 64-Bit Technology

AMD x86-64 Architecture

Programmer's Manual

Volume 3:

**General-Purpose and System
Instructions**

| Publication No. | Revision | Date |
|-----------------|----------|-----------|
| 24594-1 | 1.85 | June 2002 |

© 2002 Advanced Micro Devices, Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks

AMD, the AMD logo, AMD Athlon, AMD Duron, AMD-K5, 3DNow!, and combinations thereof, and Am486 and Am5x86 are trademarks, and AMD-K6 is a registered trademark, of Advanced Micro Devices, Inc.

MMX is a trademark and Pentium is a registered trademark of Intel Corporation.

Windows NT is a trademark of Microsoft Corp.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

| | |
|---|-------------|
| Figures | ix |
| Tables | xi |
| Preface | xiii |
| About This Book | .xiii |
| Audience | .xiii |
| Organization | .xiii |
| Definitions | .xiv |
| Related Documents | .xxv |
| 1 Instruction Formats | 1 |
| 1.1 Instruction Byte Order | 1 |
| 1.2 Instruction Prefixes | 3 |
| Summary of Legacy Prefixes | 3 |
| Operand-Size Override Prefix | 5 |
| Address-Size Override Prefix | 6 |
| Segment-Override Prefixes | 9 |
| Lock Prefix | 10 |
| Repeat Prefixes | 10 |
| REX Prefixes | 14 |
| 1.3 Opcode | 19 |
| 1.4 ModRM and SIB Bytes | 20 |
| 1.5 Displacement Bytes | 22 |
| 1.6 Immediate Bytes | 22 |
| 1.7 RIP-Relative Addressing | 23 |
| Encoding | 23 |
| REX Prefix and RIP-Relative Addressing | 24 |
| Address-Size Prefix and RIP-Relative Addressing | 24 |
| 2 Instruction Overview | 25 |
| 2.1 Instruction Subsets | 25 |
| 2.2 Reference-Page Format | 26 |
| 2.3 Summary of Registers and Data Types | 28 |
| General-Purpose Instructions | 28 |
| System Instructions | 31 |
| 128-Bit Media Instructions | 33 |
| 64-Bit Media Instructions | 36 |
| x87 Floating-Point Instructions | 38 |
| 2.4 Summary of Exceptions | 39 |
| 2.5 Notation | 40 |
| Mnemonic Syntax | 40 |
| Opcode Syntax | 44 |
| Pseudocode Definitions | 46 |

3 General-Purpose Instruction Reference 57

| | |
|--------------------|-----|
| AAA | 58 |
| AAD | 59 |
| AAM | 61 |
| AAS | 62 |
| ADC | 63 |
| ADD | 66 |
| AND | 68 |
| BOUND | 71 |
| BSF | 73 |
| BSR | 75 |
| BSWAP | 76 |
| BT | 77 |
| BTC | 79 |
| BTR | 81 |
| BTS | 83 |
| CALL | 85 |
| CBW | 92 |
| CDQ | 93 |
| CDQE | 94 |
| CLC | 95 |
| CLD | 96 |
| CLFLUSH | 97 |
| CMC | 99 |
| CMOV _{cc} | 100 |
| CMP | 104 |
| CMPS _x | 107 |
| CMPXCHG | 110 |
| CMPXCHG8B | 112 |
| CPUID | 114 |
| CQO | 130 |
| CWD | 131 |
| CWDE | 132 |
| DAA | 133 |
| DAS | 134 |
| DEC | 135 |
| DIV | 137 |
| ENTER | 139 |
| IDIV | 141 |
| IMUL | 143 |
| IN | 146 |
| INC | 147 |
| INS _x | 149 |
| INT | 152 |
| INTO | 159 |
| Jcc | 162 |
| JCXZ | 166 |

| | |
|---------------|-----|
| JMP | 167 |
| LAHF | 173 |
| LDS | 174 |
| LEA | 176 |
| LEAVE | 178 |
| LES | 180 |
| LFENCE | 182 |
| LFS | 183 |
| LGS | 185 |
| LODSx | 187 |
| LOOPcc | 189 |
| LSS | 191 |
| MFENCE | 193 |
| MOV | 194 |
| MOVD | 198 |
| MOVMSKPD | 201 |
| MOVMSKPS | 203 |
| MOVNTI | 205 |
| MOVsx | 206 |
| MOVsx | 209 |
| MOVsxD | 210 |
| MOVZX | 211 |
| MUL | 213 |
| NEG | 215 |
| NOP | 217 |
| NOT | 218 |
| OR | 220 |
| OUT | 223 |
| OUTsx | 224 |
| POP | 226 |
| POPAsx | 229 |
| POPfx | 231 |
| PREFETCHx | 234 |
| PREFETCHlevel | 236 |
| PUSH | 238 |
| PUSHAsx | 240 |
| PUSHFx | 241 |
| RCL | 243 |
| RCR | 245 |
| RET | 247 |
| ROL | 251 |
| ROR | 253 |
| SAHF | 255 |
| SAL | 256 |
| SAR | 258 |
| SBB | 261 |
| SCASx | 264 |

| | | |
|----------|-------------------------------------|------------|
| | SETcc | 267 |
| | SFENCE | 269 |
| | SHL | 270 |
| | SHLD | 272 |
| | SHR | 275 |
| | SHRD | 277 |
| | STC | 280 |
| | STD | 281 |
| | STOSx | 282 |
| | SUB | 284 |
| | TEST | 287 |
| | XADD | 289 |
| | XCHG | 291 |
| | XLATx | 293 |
| | XOR | 295 |
| 4 | System Instruction Reference | 299 |
| | ARPL | 300 |
| | CLI | 302 |
| | CLTS | 304 |
| | HLT | 305 |
| | INT 3 | 306 |
| | INVD | 312 |
| | INVLPG | 313 |
| | IRETx | 314 |
| | LAR | 320 |
| | LGDT | 323 |
| | LIDT | 325 |
| | LLDT | 327 |
| | LMSW | 329 |
| | LSL | 330 |
| | LTR | 333 |
| | MOV(CRn) | 335 |
| | MOV(DRn) | 337 |
| | RDMSR | 339 |
| | RDPMC | 340 |
| | RDTSC | 341 |
| | RSM | 342 |
| | SGDT | 343 |
| | SIDT | 345 |
| | SLDT | 347 |
| | SMSW | 349 |
| | STI | 351 |
| | STR | 353 |
| | SWAPGS | 355 |
| | SYSCALL | 357 |
| | SYSENTER | 362 |
| | SYSEXIT | 364 |

| | | |
|-------------------|---|------------|
| | SYSRET | 366 |
| | UD2 | 370 |
| | VERR | 371 |
| | VERW | 373 |
| | WBINVD | 375 |
| | WRMSR | 376 |
| Appendix A | Opcode and Operand Encodings | 379 |
| | A.1 Opcode-Syntax Notation | 379 |
| | A.2 Opcode Encodings | 381 |
| | One-Byte Opcodes | 381 |
| | Two-Byte Opcodes | 384 |
| | rFLAGS Condition Codes for Two-Byte Opcodes | 393 |
| | ModRM Extensions to One-Byte and Two-Byte Opcodes. . . | 394 |
| | ModRM Extensions to SWAPGS and CLFLUSH Opcodes . . | 397 |
| | 3DNow!™ Opcodes | 397 |
| | x87 Encodings | 400 |
| | rFLAGS Condition Codes for x87 Opcodes | 409 |
| | A.3 Operand Encodings | 409 |
| | ModRM Operand References | 409 |
| | SIB Operand References | 415 |
| Appendix B | General-Purpose Instructions in 64-Bit Mode | 419 |
| | B.1 General Rules for 64-Bit Mode | 419 |
| | B.2 Operation and Operand Size in 64-Bit Mode | 420 |
| | B.3 Invalid and Reassigned Instructions in 64-Bit Mode | 450 |
| | B.4 Instructions with 64-Bit Default Operand Size | 452 |
| | B.5 Single-Byte INC and DEC Instructions in 64-Bit Mode . . . | 454 |
| | B.6 NOP in 64-Bit Mode | 454 |
| | B.7 Segment Override Prefixes in 64-Bit Mode | 455 |
| Appendix C | Differences between Long Mode and Legacy Mode | 457 |
| Appendix D | Instruction Subsets and CPUID Feature Sets | 459 |
| | D.1 Instruction Subsets | 459 |
| | D.2 CPUID Feature Sets | 461 |
| | D.3 Instruction List | 463 |
| Appendix E | Instruction Effects on RFLAGS | 503 |
| | Index | 509 |

Figures

| | |
|--|-----|
| Figure 1-1. Instruction Byte-Order | 1 |
| Figure 1-2. Little-Endian Byte-Order of Instruction Stored in Memory . . . | 2 |
| Figure 1-3. Encoding Examples of REX-Prefix R, X, and B Bits. | 18 |
| Figure 1-4. ModRM-Byte Format | 21 |
| Figure 1-5. SIB-Byte Format | 21 |
| Figure 2-1. Format of Instruction-Detail Pages | 27 |
| Figure 2-2. General Registers in Legacy and Compatibility Modes. | 28 |
| Figure 2-3. General Registers in 64-Bit Mode. | 29 |
| Figure 2-4. Segment Registers | 30 |
| Figure 2-5. General-Purpose Data Types. | 31 |
| Figure 2-6. System Registers | 32 |
| Figure 2-7. System Data Structures | 33 |
| Figure 2-8. 128-Bit Media Registers. | 34 |
| Figure 2-9. 128-Bit Media Data Types | 35 |
| Figure 2-10. 64-Bit Media Registers. | 36 |
| Figure 2-11. 64-Bit Media Data Types | 37 |
| Figure 2-12. x87 Registers. | 38 |
| Figure 2-13. x87 Data Types | 39 |
| Figure 2-14. Syntax for Typical Two-Operand Instruction | 41 |
| Figure 3-1. Processor Signature (EAX Register) | 116 |
| Figure 3-2. Initial APIC ID, CLFLUSH Size, and Brand ID (EBX Register) . | 118 |
| Figure 3-3. Advanced Power Management Features (EDX Register) . . . | 128 |
| Figure 4-1. ModRM-Byte <i>reg</i> Field | 394 |
| Figure 4-2. ModRM-Byte Format | 410 |
| Figure 4-3. SIB-Byte Format | 416 |
| Figure D-1. Instruction Subsets vs. CPUID Feature Sets | 460 |

Tables

| | | |
|-------------|--|-----|
| Table 1-1. | Legacy Instruction Prefixes | 4 |
| Table 1-2. | Operand-Size Overrides..... | 5 |
| Table 1-3. | Address-Size Overrides | 7 |
| Table 1-4. | Pointer and Count Registers and the Address-Size Prefix | 8 |
| Table 1-5. | Segment-Override Prefixes..... | 9 |
| Table 1-6. | REP Prefix Opcodes..... | 11 |
| Table 1-7. | REPE and REPZ Prefix Opcodes | 12 |
| Table 1-8. | REPNE and REPNZ Prefix Opcodes | 13 |
| Table 1-9. | REX Instruction Prefixes | 15 |
| Table 1-10. | Instructions Not Requiring REX Prefix in 64-Bit Mode..... | 15 |
| Table 1-11. | REX Prefix-Byte Fields | 16 |
| Table 1-12. | Special REX Encodings for Registers | 19 |
| Table 1-13. | Encoding for RIP-Relative Addressing | 24 |
| Table 2-1. | Interrupt-Vector Source and Cause | 39 |
| Table 2-2. | +rb, +rw, +rd, and +rq Register Value..... | 45 |
| Table 3-1. | Processor Vendor Return Values | 115 |
| Table 3-2. | Effective Family Computation | 117 |
| Table 3-3. | Effective Model Computation..... | 118 |
| Table 3-4. | CPUID Standard Feature Support (Function 1)..... | 119 |
| Table 3-5. | CPUID AMD Feature Support (Function 8000_0001h) | 122 |
| Table 3-6. | Processor Name String Example | 125 |
| Table 3-7. | CPUID TLB Bits for 2-Mbyte and 4-Mbyte Pages | 126 |
| Table 3-8. | CPUID TLB Bits for 4-Kbyte Pages | 126 |
| Table 3-9. | CPUID L1 Data Cache Bits | 126 |
| Table 3-10. | CPUID L1 Instruction Cache Bits..... | 126 |
| Table 3-11. | CPUID L2 TLB Bits for 2-Mbyte and 4-Mbyte Pages | 127 |
| Table 3-12. | CPUID L2 TLB Bits for 4-Kbyte Pages..... | 127 |
| Table 3-13. | CPUID L2 Cache Bits..... | 127 |
| Table 3-14. | CPUID Long-Mode Address Sizes..... | 129 |
| Table 3-15. | Summary of PREFETCH Instruction Type Options..... | 235 |
| Table 3-16. | Locality References for the Prefetch Instructions | 236 |
| Table 4-1. | One-Byte Opcodes, Low Nibble 0–7h | 381 |
| Table 4-2. | One-Byte Opcodes, Low Nibble 8–Fh..... | 383 |
| Table 4-3. | Second Byte of Two-Byte Opcodes, Low Nibble 0–7h..... | 384 |

| | | |
|-------------|---|-----|
| Table 4-4. | Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh | 388 |
| Table 4-5. | rFLAGS Condition Codes for CMOVcc, Jcc, and SETcc | 393 |
| Table 4-6. | One-Byte and Two-Byte Opcode ModRM Extensions | 395 |
| Table 4-7. | SWAPGS and CLFLUSH ModRM Extensions | 397 |
| Table 4-8. | Immediate Byte for 3DNow!™ Opcodes, Low Nibble 0–7h . . | 398 |
| Table 4-9. | Immediate Byte for 3DNow!™ Opcodes, Low Nibble 8–Fh . . | 399 |
| Table 4-10. | x87 Opcodes and ModRM Extensions | 401 |
| Table 4-11. | rFLAGS Condition Codes for FCMOVcc | 409 |
| Table 4-12. | ModRM Register References, 32/64-Bit Address | 411 |
| Table 4-13. | ModRM Memory References, 32/64-Bit Address | 412 |
| Table 4-14. | ModRM Register References, 16-Bit Address | 413 |
| Table 4-15. | ModRM Memory References, 16-Bit Address | 414 |
| Table 4-16. | SIB <i>base</i> Field References | 416 |
| Table 4-17. | SIB Memory References | 417 |
| Table B-1. | Operations and Operands in 64-Bit Mode | 421 |
| Table B-2. | Invalid Instructions in 64-Bit Mode | 451 |
| Table B-3. | Reassigned Instructions in 64-Bit Mode | 452 |
| Table B-4. | Invalid Instructions in Long Mode | 452 |
| Table B-5. | Instructions Defaulting to 64-Bit Operand Size | 453 |
| Table C-1. | Differences Between Long Mode and Legacy Mode | 457 |
| Table D-1. | Instruction Subsets and CPUID Feature Sets | 463 |
| Table E-1. | Instruction Effects on RFLAGS | 503 |

Preface

About This Book

This book is part of a multivolume work entitled the *AMD x86-64 Architecture Programmer's Manual*. This table lists each volume and its order number.

| Title | Order No. |
|--|-----------------------------|
| Volume 1, <i>Application Programming</i> | 24592 |
| Volume 2, <i>System Programming</i> | 24593 |
| Volume 3, <i>General-Purpose and System Instructions</i> Volume 4, <i>128-Bit Media Instructions</i> Volume 5, <i>64-Bit Media and x87 Floating-Point Instructions</i> | 24594 (three-volume kit) |

Audience

This volume (Volume 3) is intended for all programmers writing application or system software for a processor that implements the x86-64 architecture. Descriptions of general-purpose instructions assume an understanding of the application-level programming topics described in Volume 1. Descriptions of system instructions assume an understanding of the system-level programming topics described in Volume 2.

Organization

Volumes 3, 4, and 5 describe the x86-64 architecture's instruction set in detail. Together, they cover each instruction's mnemonic syntax, opcodes, functions, affected flags, and possible exceptions.

The x86-64 instruction set is divided into five subsets:

- General-purpose instructions
- System instructions
- 128-bit media instructions
- 64-bit media instructions
- x87 floating-point instructions

Several instructions belong to—and are described identically in—multiple instruction subsets.

This volume describes the general-purpose and system instructions. The index at the end cross-references topics within this volume. For other topics relating to the x86-64 architecture, and for information on instructions in other subsets, see the tables of contents and indexes of the other volumes.

Definitions

Many of the following definitions assume an in-depth knowledge of the legacy x86 architecture. See “Related Documents” on page xxv for descriptions of the legacy x86 architecture.

Terms and Notation

In addition to the notation described below, “Opcode-Syntax Notation” on page 379 describes notation relating specifically to opcodes.

1011b

A binary value—in this example, a 4-bit value.

F0EAh

A hexadecimal value—in this example a 2-byte value.

[1,2)

A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).

7–4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.

128-bit media instructions

Instructions that use the 128-bit XMM registers. These are a combination of the SSE and SSE2 instruction sets.

64-bit media instructions

Instructions that use the 64-bit MMX™ registers. These are primarily a combination of MMX and 3DNow!™ instruction sets, with some additional instructions from the SSE and SSE2 instruction sets.

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See *legacy mode* and *compatibility mode*.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode*.

64-bit mode

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

#GP(0)

Notation indicating a general-protection exception (#GP) with error code of 0.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

byte

Eight bits.

clear

To write a bit value of 0. Compare *set*.

compatibility mode

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction's result to software-visible storage, such as a register

(including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

CR0–CR4

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0.PE = 1

Notation indicating that the PE bit of the CR0 register has a value of 1.

direct

Referencing a memory location whose address is included in the instruction's syntax as an immediate operand. The address may be an absolute or relative address. Compare *indirect*.

dirty data

Data held in the processor's caches or internal buffers that is more recent than the copy held in main memory.

displacement

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

doubleword

Two words, or four bytes, or 32 bits.

double quadword

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

DS:rSI

The contents of a memory location whose segment address is in the DS register and whose offset relative to that segment is in the rSI register.

EFER.LME = 0

Notation indicating that the LME bit of the EFER register has a value of 0.

effective address size

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

effective operand size

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

element

See *vector*.

exception

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. For all exceptions except 128-bit media SIMD floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

FF /0

Notation indicating that FF is the first byte of an opcode, and a subfield in the second byte has a value of 0.

flush

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

GDT

Global descriptor table.

IDT

Interrupt descriptor table.

IGN

Ignore. Field is ignored.

indirect

Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare *direct*.

IRB

The virtual-8086 mode interrupt-redirection bitmap.

IST

The long-mode interrupt-stack table.

IVT

The real-address mode interrupt-vector table.

LDT

Local descriptor table.

legacy x86

The legacy x86 architecture. See “Related Documents” on page xxv for descriptions of the legacy x86 architecture.

legacy mode

An operating mode of the x86-64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the x86-64 architecture can run in either *long mode* or *legacy mode*. Legacy mode has three submodes, *real mode*, *protected mode*, and *virtual-8086 mode*.

long mode

An operating mode unique to the x86-64 architecture. A processor implementation of the x86-64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

lsb

Least-significant bit.

LSB

Least-significant byte.

main memory

Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask

(1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ

Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs.

memory

Unless otherwise specified, *main memory*.

ModRM

A byte following an instruction opcode that specifies address calculation based on mode (Mod), register (R), and memory (M) variables.

moffset

A direct memory offset. In other words, a displacement that is added to the base of a code segment (for absolute addressing) or to an instruction pointer (for addressing relative to the instruction pointer, as in RIP-relative addressing).

msb

Most-significant bit.

MSB

Most-significant byte.

multimedia instructions

A combination of *128-bit media instructions* and *64-bit media instructions*.

octword

Same as *double quadword*.

offset

Same as *displacement*.

overflow

The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.

packed

See *vector*.

PAE

Physical-address extensions.

physical memory

Actual memory, consisting of *main memory* and cache.

probe

A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

protected mode

A submode of *legacy mode*.

quadword

Four words, or eight bytes, or 64 bits.

RAZ

Read as zero (0), regardless of what is written.

real-address mode

See *real mode*.

real mode

A short name for *real-address mode*, a submode of *legacy mode*.

relative

Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

REX

An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing

Addressing relative to the 64-bit RIP instruction pointer. Compare *moffset*.

set

To write a bit value of 1. Compare *clear*.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SIMD

Single instruction, multiple data. See *vector*.

SSE

Streaming SIMD extensions instruction set. See *128-bit media instructions* and *64-bit media instructions*.

SSE2

Extensions to the SSE instruction set. See *128-bit media instructions* and *64-bit media instructions*.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

TOP

The x87 top-of-stack pointer.

TPR

Task-priority register (CR8).

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called *elements*, that are packed into a single operand. Most of the 128-bit and 64-bit media instructions use vectors as operands. Vectors are also called *packed* or *SIMD* (single-instruction multiple-data) operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

virtual-8086 mode

A submode of *legacy mode*.

word

Two bytes, or 16 bits.

x86

See *legacy x86*.

Registers

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

AH–DH

The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

AL–DL

The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

AL–r15B

The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

BP

Base pointer register.

CR_n

Control register number *n*.

CS

Code segment register.

eAX–eSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

EBP

Extended base pointer register.

EFER

Extended features enable register.

eFLAGS

16-bit or 32-bit flags register. Compare *rFLAGS*.

EFLAGS

32-bit (extended) flags register.

eIP

16-bit or 32-bit instruction-pointer register. Compare *rIP*.

EIP

32-bit (extended) instruction-pointer register.

FLAGS

16-bit flags register.

GDTR

Global descriptor table register.

GPRs

General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR

Interrupt descriptor table register.

IP

16-bit instruction-pointer register.

LDTR

Local descriptor table register.

MSR

Model-specific register.

r8–r15

The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with

nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

RAX

64-bit version of the EAX register.

RBP

64-bit version of the EBP register.

RBX

64-bit version of the EBX register.

RCX

64-bit version of the ECX register.

RDI

64-bit version of the EDI register.

RDX

64-bit version of the EDX register.

rFLAGS

16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

RFLAGS

64-bit flags register. Compare *rFLAGS*.

rIP

16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

RIP

64-bit instruction-pointer register.

RSI

64-bit version of the ESI register.

RSP

64-bit version of the ESP register.

SP

Stack pointer register.

SS

Stack segment register.

TPR

Task priority register, a new register introduced in the x86-64 architecture to speed interrupt management.

TR

Task register.

Endian Order

The x86 and x86-64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD, *AMD-K6™ MMX™ Enhanced Processor Multimedia Technology*, Sunnyvale, CA, 2000.
- AMD, *3DNow!™ Technology Manual*, Sunnyvale, CA, 2000.
- AMD, *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets*, Sunnyvale, CA, 2000.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.
- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts*, Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals*, Addison-Wesley, New York, 1994.

- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
- Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
- Jeffrey P. Doyer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC*, Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.
- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2 Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightning 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.

- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book*, Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*, www.x86.org/articles/sse_pt1/simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible*, Microsoft Press, Redmond, WA, 1993.
- *PharLap 386/ASM Reference Manual*, Pharlap, Cambridge MA, 1993.
- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.

- Web sites and newsgroups:
 - www.amd.com
 - news.comp.arch
 - news.comp.lang.asm.x86
 - news.intel.microprocessors
 - news.microsoft

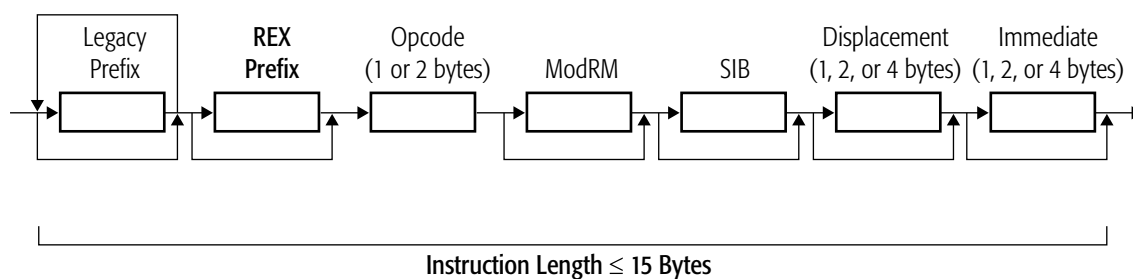
1 Instruction Formats

An instruction's format encodes its operation, as well as encoding the locations of its initial operands and the result of the operation. This section describes the general format and parameters used by all instructions. For information on the specific format(s) for each instruction, see:

- Chapter 3, “General-Purpose Instruction Reference.”
- Chapter 4, “System Instruction Reference.”
- “128-Bit Media Instruction Reference” in Volume 4.
- “64-Bit Media Instruction Reference” in Volume 5.
- “x87 Floating-Point Instruction Reference” in Volume 5.

1.1 Instruction Byte Order

An instruction can be between one and 15 bytes in length. Figure 1-1 shows the byte order of the instruction format.



513-303.eps

Figure 1-1. Instruction Byte-Order

Instructions are stored in memory in little-endian order. The least-significant byte of an instruction is stored at its lowest memory address, as shown in Figure 1-2.

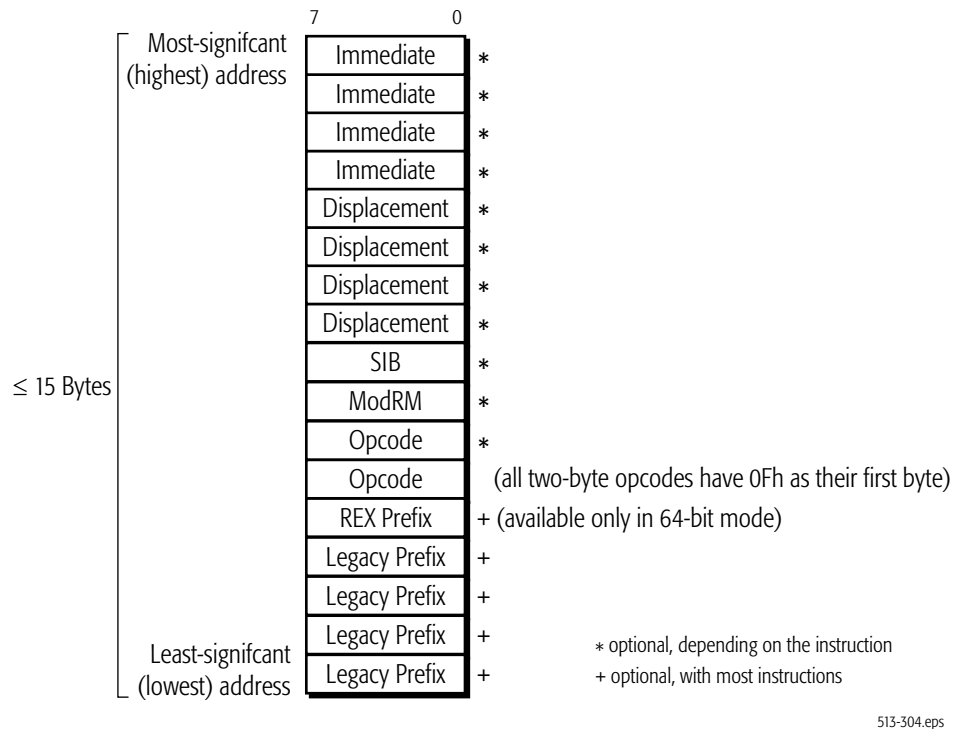


Figure 1-2. Little-Endian Byte-Order of Instruction Stored in Memory

The basic operation of an instruction is specified by an *opcode*. The opcode is one or two bytes long, as described in “Opcode” on page 19. An opcode can be preceded by up to four *legacy prefixes*—one from each of four groups of prefixes, as described in “Instruction Prefixes” on page 3. The legacy prefixes modify an instruction’s default address size, operand size, or segment, or they invoke a special function such as modification of the opcode, atomic bus-locking, or repetition. The *REX prefix* can be used in 64-bit mode to access the register extensions illustrated in “Application-Programming Register Set” in Volume 1. If a REX prefix is used, it must immediately precede the first opcode byte.

An instruction’s opcode consists of one or two bytes. In several 128-bit and 64-bit media instructions, a legacy operand-size or repeat prefix byte is used in a special-purpose way to modify the opcode. The opcode can be followed by a *mode-register-memory (ModRM) byte*, which further describes the operation and/or operands. The opcode, or the opcode and ModRM byte,

can also be followed by a *scale-index-base (SIB) byte*, which describes the scale, index, and base forms of memory addressing. The ModRM and SIB bytes are described in “ModRM and SIB Bytes” on page 20, but their legacy functions can be modified by the REX prefix (“Instruction Prefixes” on page 3).

The 15-byte instruction-length limit can only be exceeded by using non-redundant prefixes. If the limit is exceeded, a general-protection exception occurs.

1.2 Instruction Prefixes

The instruction prefixes shown in Figure 1-1 are of two types: legacy prefixes and REX prefixes. Each of the legacy prefixes has a unique byte value. By contrast, the REX prefixes, which enable use of the x86-64 register extensions in 64-bit mode, are organized as a group of byte values in which the value of the prefix indicates the combination of register-extension features to be enabled.

1.2.1 Summary of Legacy Prefixes

Table 1-1 shows the legacy prefixes—that is, all prefixes except the REX prefixes, which are described on page 14. The legacy prefixes are organized into four groups, as shown in the left-most column of Table 1-1. A maximum of one prefix from each of the four groups can be used in a single instruction. The legacy prefixes can appear in any order within the position shown in Figure 1-1 for legacy prefixes. The result of using multiple prefixes from a single group is unpredictable.

Some of the restrictions on legacy prefixes are:

- *Operand-Size Override*—This prefix affects only general-purpose instructions and a few x87 instructions. When used with 128-bit and 64-bit media instructions, this prefix acts in a special way to modify the opcode.
- *Address-Size Override*—This prefix affects only memory operands.
- *Segment Override*—In 64-bit mode, the CS, DS, ES, and SS segment override prefixes are ignored.
- *LOCK Prefix*—This prefix affects only certain instructions that reference memory.
- *Repeat Prefixes*—These prefixes affect only certain string instructions. When used with 128-bit and 64-bit media

instructions, two of the three prefixes act in a special way to modify the opcode.

Table 1-1. Legacy Instruction Prefixes

| Prefix Group ⁵ | Mnemonic | Prefix Byte (Hex) | Description |
|---------------------------|----------------|-------------------|--|
| Operand-Size Override | none | 66 ¹ | Changes the default operand size of a memory or register operand, as shown in Table 1-2 on page 5. |
| Address-Size Override | none | 67 ² | Changes the default address size of a memory operand, as shown in Table 1-3 on page 7. |
| Segment Override | CS | 2E ⁶ | Forces use of the current CS segment for memory operands. |
| | DS | 3E ⁶ | Forces use of the current DS segment for memory operands. |
| | ES | 26 ⁶ | Forces use of the current ES segment for memory operands. |
| | FS | 64 | Forces use of the current FS segment for memory operands. |
| | GS | 65 | Forces use of the current GS segment for memory operands. |
| | SS | 36 ⁶ | Forces use of the current SS segment for memory operands. |
| Lock or Repeat | LOCK | F0 ⁴ | Causes certain kinds of memory read-modify-write instructions to occur atomically. |
| | REP | F3 ³ | Repeats a string operation (INS, MOVS, OUTS, LODS, and STOS) until the rCX register equals 0. |
| | REPE or REPZ | | Repeats a compare-string or scan-string operation (CMPSx and SCASx) until the rCX register equals 0 or the zero flag (ZF) is cleared to 0. |
| | REPNE or REPNZ | F2 ³ | Repeats a compare-string or scan-string operation (CMPSx and SCASx) until the rCX register equals 0 or the zero flag (ZF) is set to 1. |

Note:

1. When used with 128-bit and 64-bit media instructions, this prefix acts in a special way to modify the opcode. The prefix is ignored by 64-bit media floating-point (3DNow!) instructions. See "Instructions that Cannot Use the Operand-Size Prefix" on page 6.
2. This prefix changes address size only for a memory operand, and only a single memory operand can be overridden in a given instruction. The prefix is ignored by instructions that have no memory operand.
3. This prefix should be used only with compare-string and scan-string instructions. When used with 128-bit and 64-bit media instructions, the prefix acts in a special way to modify the opcode.
4. The LOCK prefix should not be used for instructions other than those listed in "Lock Prefix" on page 10.
5. A maximum of one prefix from each of the four groups can be used with a single instruction.
6. In 64-bit mode, the CS, DS, ES, and SS segment overrides are ignored.

1.2.2 Operand-Size Override Prefix

The default operand size for an instruction is determined by a combination of its opcode, the D (default) bit in the current code-segment descriptor, and the current operating mode, as shown in Table 1-2. The operand-size override prefix (66h) selects the non-default operand size. The prefix can be used with any general-purpose instruction that accesses non-fixed-size operands in memory or general-purpose registers (GPRs), and it can also be used with the x87 FLDENV, FNSTENV, FNSAVE, and FRSTOR instructions.

In 64-bit mode, the prefix allows mixing of 16-bit, 32-bit, and 64-bit data on an instruction-by-instruction basis. In compatibility and legacy modes, the prefix allows mixing of 16-bit and 32-bit operands on an instruction-by-instruction basis.

Table 1-2. Operand-Size Overrides

| Operating Mode | | Default Operand Size (Bits) | Effective Operand Size (Bits) | Instruction Prefix ¹ | |
|----------------|---|-----------------------------|-------------------------------|---------------------------------|------------------|
| | | | | 66h | REX ³ |
| Long Mode | 64-Bit Mode | 32 ² | 64 | don't care | yes |
| | | | 32 | no | no |
| | | | 16 | yes | no |
| | Compatibility Mode | 32 | 32 | no | Not Applicable |
| | | | 16 | yes | |
| | | 16 | 32 | yes | |
| | | | 16 | no | |
| | Legacy Mode (Protected, Virtual-8086, or Real Mode) | 32 | 32 | no | |
| 16 | | | yes | | |
| 16 | | 32 | yes | | |
| | | 16 | no | | |

Note:

1.

2.

3.

"no" indicates that the default operand size is used.

This is the typical default, although some instructions default to other operand sizes. See Appendix B, "General-Purpose Instructions in 64-Bit Mode," for details.

See "REX Prefixes" on page 14.

In 64-bit mode, most instructions default to a 32-bit operand size. For these instructions, a REX prefix (page 16) specifies a 64-bit operand size, and a 66h prefix specifies a 16-bit operand size. The REX prefix takes precedence over the 66h prefix. However, if an instruction defaults to a 64-bit operand size, it does not need a REX prefix and it can only be overridden to a 16-bit operand size. It cannot be overridden to a 32-bit operand size, because there is no 32-bit operand-size override prefix in 64-bit mode. Two groups of instructions have a default 64-bit operand size in 64-bit mode:

- Near branches. For details, see “Near Branches in 64-Bit Mode” in Volume 1.
- All instructions, except far branches, that implicitly reference the RSP. For details, see “Stack Operation” in Volume 1.

Instructions that Cannot Use the Operand-Size Prefix. The operand-size prefix should be used only with general-purpose instructions and the x87 FLDENV, FNSTENV, FNSAVE, and FRSTOR instructions, in which the prefix selects between 16-bit and 32-bit operand size. The prefix is ignored by all other x87 instructions and by 64-bit media floating-point (3DNow!) instructions.

When used with 64-bit media *integer* instructions, the 66h prefix acts in a special way to modify the opcode. This modification typically causes an access to an XMM register or 128-bit memory operand and thereby converts the 64-bit media instruction into its comparable 128-bit media instruction. The result of using an F2h or F3h repeat prefix along with a 66h prefix in 128-bit or 64-bit media instructions is unpredictable.

Operand-Size and REX Prefixes. For non-byte operations, the REX operand-size prefix takes precedence over the 66h prefix. See “REX.W: Operand Width” on page 16 for details.

1.2.3 Address-Size Override Prefix

The default address size for instructions that access non-stack memory is determined by the current operating mode, as shown in Table 1-3. The address-size override prefix (67h) selects the non-default address size. Depending on the operating mode, this prefix allows mixing of 16-bit and 32-bit, or of 32-bit and 64-bit addresses, on an instruction-by-instruction basis. The prefix changes the address size only for memory operands and is thus meaningful only with instructions that access operands in

memory. It is ignored by instructions that have no memory operands.

For instructions that access a stack segment (SS), the default address size for stack accesses is determined by the D (default) bit in the stack-segment descriptor. In 64-bit mode, the D bit is ignored, and all stack references have a 64-bit address size. However, if an instruction accesses both stack and non-stack memory, the address size of the non-stack access is determined as shown in Table 1-3.

Table 1-3. Address-Size Overrides

| Operating Mode | | Default Address Size (Bits) | Effective Address Size (Bits) | Address-Size Prefix (67h) ¹ Required? |
|---|--------------------|-----------------------------|-------------------------------|--|
| Long Mode | 64-Bit Mode | 64 | 64 | no |
| | | | 32 | yes |
| | Compatibility Mode | 32 | 32 | no |
| | | | 16 | yes |
| | | 16 | 32 | yes |
| | | | 16 | no |
| Legacy Mode (Protected, Virtual-8086, or Real Mode) | | 32 | 32 | no |
| | | | 16 | yes |
| | | 16 | 32 | yes |
| | | | 16 | no |

Note:

1. “no” indicates that the default address size is used.

As Table 1-3 shows, the default address size is 64 bits in 64-bit mode. The size can be overridden to 32 bits, but 16-bit addresses are not supported in 64-bit mode. In compatibility and legacy modes, the default address size is 16 bits or 32 bits, depending on the operating mode (see "Processor Initialization and Long-Mode Activation" in Volume 2 for details). In these modes, the address-size prefix selects the non-default size, but the 64-bit address size is not available.

Certain instructions reference pointer registers or count registers implicitly, rather than explicitly. In such instructions, the address-size prefix affects the size of such addressing and count registers, just as it does when such registers are explicitly referenced. Table 1-3 lists all such instructions and the registers referenced using the three possible address sizes.

Table 1-4. Pointer and Count Registers and the Address-Size Prefix

| Instruction | Pointer or Count Register | | |
|--|---------------------------|---------------------|---------------------|
| | 16-Bit Address Size | 32-Bit Address Size | 64-Bit Address Size |
| CMPS, CMPSB, CMPSW, CMPSD, CMPSQ —Compare Strings | SI, DI | ESI, EDI | RSI, RDI |
| INS, INSB, INSD, INSW —Input String | DI | EDI | RDI |
| JCXZ, JECXZ, JRCXZ —Jump on CX/ECX/RCX Zero | CX | ECX | RCX |
| LODS, LODSB, LODSW, LODSD, LODSQ —Load String | SI | ESI | RSI |
| LOOP, LOOPE, LOOPNZ, LOOPNE, LOOPZ —Loop | CX | ECX | RCX |
| MOVS, MOVSB, MOVSW, MOVSD, MOVSQ —Move String | SI, DI | ESI, EDI | RSI, RDI |
| OUTS, OUTSB, OUTSD, OUTSW —Output String | SI | ESI | RSI |
| REP, REPE, REPNE, REPNZ, REPZ —Repeat Prefixes | CX | ECX | RCX |
| SCAS, SCASB, SCASW, SCASD, SCASQ —Scan String | DI | EDI | RDI |
| STOS, STOSB, STOSW, STOSD, STOSQ —Store String | DI | EDI | RDI |
| XLAT, XLATB —Table Look-up Translation | BX | EBX | RBX |

1.2.4 Segment-Override Prefixes

Segment overrides can be used only with instructions that reference non-stack memory. Most instructions that reference memory are encoded with a ModRM byte (page 20). The default segment for such memory-referencing instructions is implied by the base register indicated in its ModRM byte, as follows:

- *Instructions that Reference a Non-Stack Segment*—If an instruction encoding references any base register other than rBP or rSP, or if an instruction contains an immediate offset, the default segment is the data segment (DS). These instructions can use the segment-override prefix to select one of the non-default segments, as shown in Table 1-5. For details on operating in 64-bit mode, see “Segment Overrides in 64-Bit Mode” on page 10.
- *String Instructions*—String instructions reference two memory operands. By default, they reference both the DS and ES segments (DS:rSI and ES:rDI). In legacy and compatibility modes, these instructions can override their DS-segment reference, as shown in Table 1-5, but they cannot override their ES-segment reference. For details on operating in 64-bit mode, see “Segment Overrides in 64-Bit Mode” on page 10.
- *Instructions that Reference the Stack Segment*—If an instruction’s encoding references the rBP or rSP base register, the default segment is the stack segment (SS). All instructions that reference the stack (push, pop, call, interrupt, return from interrupt) use SS by default. These instructions cannot use the segment-override prefix.

Table 1-5. Segment-Override Prefixes

| Mnemonic | Prefix Byte (Hex) | Description |
|--|-------------------|---|
| CS ¹ | 2E | Forces use of current CS segment for memory operands. |
| DS ¹ | 3E | Forces use of current DS segment for memory operands. |
| ES ¹ | 26 | Forces use of current ES segment for memory operands. |
| FS | 64 | Forces use of current FS segment for memory operands. |
| Note: 1. In 64-bit mode, the CS, DS, ES, and SS segment overrides are ignored. | | |

Table 1-5. Segment-Override Prefixes (continued)

| Mnemonic | Prefix Byte (Hex) | Description |
|--|-------------------|---|
| GS | 65 | Forces use of current GS segment for memory operands. |
| SS ¹ | 36 | Forces use of current SS segment for memory operands. |
| Note: 1. In 64-bit mode, the CS, DS, ES, and SS segment overrides are ignored. | | |

Segment Overrides in 64-Bit Mode. In 64-bit mode, the CS, DS, ES, and SS segment-override prefixes have no effect. These four prefixes are not treated as segment-override prefixes for the purposes of the multiple-prefix (maximum of four) rule. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as true segment-override prefixes in 64-bit mode. Use of the FS or GS prefix causes their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” in Volume 2 for details.

1.2.5 Lock Prefix

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

1.2.6 Repeat Prefixes

The repeat prefixes cause repetition of certain instructions that load, store, move, input, or output strings. The prefixes should only be used with such string instructions. Two pairs of repeat prefixes, REPE/REPZ and REPNE/REPZ, perform the same repeat functions for certain compare-string and scan-string instructions. The repeat function uses rCX as a count register. The operand size of rCX is based on address size, as shown in Table 1-4 on page 8.

REP. The REP prefix repeats its associated string instruction the number of times specified in the counter register (rCX). It terminates the repetition when the value in rCX reaches 0. The prefix can only be used with the INS, LODS, MOVS, OUTS, and STOS instructions. Table 1-6 shows the valid REP prefix opcodes.

Table 1-6. REP Prefix Opcodes

| Mnemonic | Opcode |
|---|--------|
| REP INS <i>reg/mem8</i> , DX REP INSB | F3 6C |
| REP INS <i>reg/mem16/32</i> , DX REP INSW REP INSD | F3 6D |
| REP LODS <i>mem8</i> REP LODSB | F3 AC |
| REP LODS <i>mem16/32/64</i> REP LODSW REP LODSD REP LODSQ | F3 AD |
| REP MOVS <i>mem8, mem8</i> REP MOVSB | F3 A4 |
| REP MOVS <i>mem16/32/64, mem16/32/64</i> REP MOVSW REP MOVSD REP MOVSQ | F3 A5 |
| REP OUTS DX, <i>reg/mem8</i> REP OUTSB | F3 6E |

Table 1-6. REP Prefix Opcodes (continued)

| Mnemonic | Opcode |
|--|--------|
| REP OUTS DX, <i>reg/mem16/32</i> REP OUTSW REP OUTSD | F3 6F |
| REP STOS <i>mem8</i> REP STOSB | F3 AA |
| REP STOS <i>mem16/32/64</i> REP STOSW REP STOSD REP STOSQ | F3 AB |

REPE and REPZ. REPE and REPZ are synonyms and have identical opcodes. These prefixes repeat their associated string instruction the number of times specified in the counter register (rCX). The repetition terminates when the value in rCX reaches 0 or when the zero flag (ZF) is cleared to 0. The REPE and REPZ prefixes can only be used with the CMPS, CMPSB, CMPSD, CMPSW, SCAS, SCASB, SCASD, and SCASW instructions. Table 1-7 shows the valid REPE and REPZ prefix opcodes.

Table 1-7. REPE and REPZ Prefix Opcodes

| Mnemonic | Opcode |
|---|--------|
| REPx CMPS <i>mem8, mem8</i> REPx CMPSB | F3 A6 |
| REPx CMPS <i>mem16/32/64, mem16/32/64</i> REPx CMPSW REPx CMPSD REPx CMPSQ | F3 A7 |
| REPx SCAS <i>mem8</i> REPx SCASB | F3 AE |
| REPx SCAS <i>mem16/32/64</i> REPx SCASW REPx SCASD REPx SCASQ | F3 AF |

REPNE and REPZ. REPNE and REPZ are synonyms and have identical opcodes. These prefixes repeat their associated string instruction the number of times specified in the counter register (rCX). The repetition terminates when the value in rCX reaches 0 or when the zero flag (ZF) is set to 1. The REPNE and REPZ prefixes can only be used with the CMPS, CMPSB, CMPSD, CMPSW, SCAS, SCASB, SCASD, and SCASW instructions. Table 1-8 shows the valid REPNE and REPZ prefix opcodes.

Table 1-8. REPNE and REPZ Prefix Opcodes

| Mnemonic | Opcode |
|---|--------|
| REPNe CMPS <i>mem8, mem8</i> REPNe CMPSB | F2 A6 |
| REPNe CMPS <i>mem16/32/64, mem16/32/64</i> REPNe CMPSW REPNe CMPSD REPNe CMPSQ | F2 A7 |
| REPNe SCAS <i>mem8</i> REPNe SCASB | F2 AE |
| REPNe SCAS <i>mem16/32/64</i> REPNe SCASW REPNe SCASD REPNe SCASQ | F2 AF |

Instructions that Cannot Use Repeat Prefixes. The repeat prefixes should only be used in the string instructions listed in tables 1-6, 1-7, and 1-8. When used in 128-bit or 64-bit media instructions, the F2h and F3h prefixes act in a special way to modify the opcode rather than cause a repeat operation. The result of using a 66h operand-size prefix along with an F2h or F3h prefix in 128-bit or 64-bit media instructions is unpredictable.

Optimization of Repeats. Depending on the hardware implementation, the repeat prefixes can have a setup overhead. If the repeated count is variable, the overhead can sometimes be avoided by substituting a simple loop to move or store the data. Repeated string instructions can be expanded into equivalent

sequences of inline loads and stores or a sequence of stores can be used to emulate a REP STOS.

For repeated string moves, performance can be maximized by moving the largest possible operand size. For example, use REP MOVSD rather than REP MOVSW and REP MOVSW rather than REP MOVSB. Use REP STOSD rather than REP STOSW and REP STOSW rather than REP MOVSB.

Depending on the hardware implementation, string moves with the direction flag (DF) cleared to 0 (up) may be faster than string moves with DF set to 1 (down). DF = 1 is only needed for certain cases of overlapping REP MOVSB, such as when the source and the destination overlap.

1.2.7 REX Prefixes

REX prefixes are a group of instruction-prefix bytes that can be used only in 64-bit mode. They enable access to the x86-64 register extensions. Figure 1-1 on page 1 and Figure 1-2 on page 2 show how a REX prefix fits within the byte order of instructions. REX prefixes enable the following features in 64-bit mode:

- Use of the extended GPR (Figure 2-3 on page 29) or XMM registers (Figure 2-8 on page 34).
- Use of the 64-bit operand size when accessing GPRs.
- Use of RIP-relative addressing, as described in “RIP-Relative Addressing” in Volume 1.
- Use of the extended control and debug registers, as described in “64-Bit-Mode Extended Control Registers” in Volume 2 and “64-Bit-Mode Extended Debug Registers” in Volume 2.

Table 1-9 shows the REX prefixes. The value of a REX prefix is in the range 40h through 4Fh, depending on the particular combination of x86-64 register extensions desired.

Table 1-9. REX Instruction Prefixes

| Prefix Type | Mnemonic | Prefix Code (Hex) | Description |
|---|----------|---|--------------------------------------|
| Register Extensions | REX.W | 40 ¹ through 4F ¹ | Access an x86-64 register extension. |
| | REX.R | | |
| | REX.X | | |
| | REX.B | | |
| Note: 1. See Table 1-11 for encoding of REX prefixes. | | | |

A REX prefix is normally required with an instruction that accesses a 64-bit GPR or one of the extended GPR or XMM registers. Only a few instructions have an operand size that defaults to (or is fixed at) 64 bits in 64-bit mode, and thus do not need a REX prefix. These exceptions to the normal rule are listed in Table 1-10.

Table 1-10. Instructions Not Requiring REX Prefix in 64-Bit Mode

| | |
|--------------------|------------------------------|
| CALL near | LOOPcc |
| CMPXCHG8B | MOV to/from Control Register |
| Jcc | MOV to/from Debug Register |
| JCXZ, JECXZ, JRCXZ | RET |
| JMP near | |

An instruction can have only one REX prefix, although the prefix can express several extension features. If a REX prefix is used, it must immediately precede the first opcode byte in the instruction format. Any other placement of a REX prefix, or any use of a REX prefix in an instruction that does not access an extended register, is ignored. The legacy instruction-size limit of 15 bytes still applies to instructions that contain a REX prefix.

REX prefixes are a set of sixteen values that span one row of the main opcode map and occupy entries 40h through 4Fh. Table 1-11 and Figure 1-3 on page 18 show the prefix fields and their uses.

Table 1-11. REX Prefix-Byte Fields

| Mnemonic | Bit Position | Definition |
|--|--------------|--|
| — | 7–4 | 0100 |
| REX.W | 3 | 0 = Default operand size 1 = 64-bit operand size |
| REX.R | 2 | 1-bit (high) extension of the ModRM <i>reg</i> field ¹ , thus permitting access to 16 registers. |
| REX.X | 1 | 1-bit (high) extension of the SIB <i>index</i> field ¹ , thus permitting access to 16 registers. |
| REX.B | 0 | 1-bit (high) extension of the ModRM <i>r/m</i> field ¹ , SIB <i>base</i> field ¹ , or opcode <i>reg</i> field, thus permitting access to 16 registers. |
| Note: 1. For a description of the ModRM and SIB bytes, see “ModRM and SIB Bytes” on page 20. | | |

REX.W: Operand Width. Setting the REX.W bit to 1 specifies a 64-bit operand size. Like the existing 66h operand-size prefix, the REX 64-bit operand-size override has no effect on byte operations. For non-byte operations, the REX operand-size override takes precedence over the 66h prefix. If a 66h prefix is used together with a REX prefix that has the REX.W bit set to 1, the 66h prefix is ignored. However, if a 66h prefix is used together with a REX prefix that has the REX.W bit cleared to 0, the 66h prefix is not ignored and the operand size becomes 16 bits.

REX.R: Register. The REX.R bit adds a 1-bit (high) extension to the ModRM *reg* field (page 20) when that field encodes a GPR or XMM register. REX.R does not modify ModRM *reg* when that field specifies other registers or opcodes. REX.R is ignored in such cases.

REX.X: Index. The REX.X bit adds a 1-bit (high) extension to the SIB *index* field (page 20).

REX.B: Base. The REX.B bit either adds a 1-bit (high) extension to the base in the ModRM *r/m* field or SIB *base* field, or it adds a 1-bit (high) extension to the opcode *reg* field used for accessing GPRs, control registers, or debug registers. (See Table 2-2 on page 45 for more about the REX.B bit.)

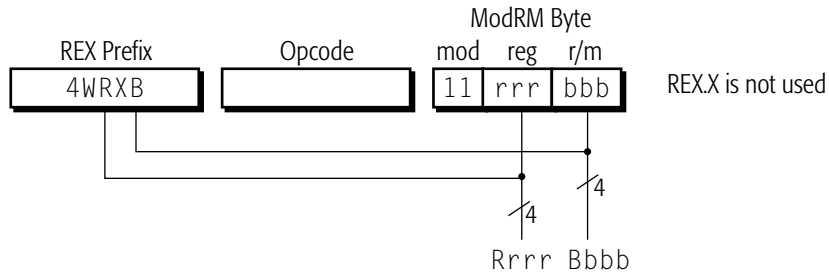
Encoding Examples. Figure 1-3 shows four examples of how the R, X, and B bits of REX prefixes are concatenated with fields from the ModRM byte, SIB byte, and opcode to specify register and memory addressing. The R, X, and B bits are described in Table 1-11 on page 16.

Byte-Register Addressing. In the legacy architecture, the byte registers (AH, AL, BH, BL, CH, CL, DH, and DL, shown in Figure 2-2 on page 28) are encoded in the ModRM *reg* or *r/m* field or in the opcode *reg* field as registers 0 through 7. The REX prefix provides an additional byte-register addressing capability that makes the least-significant byte of any GPR available for byte operations (Figure 2-3 on page 29). This provides a uniform set of byte, word, doubleword, and quadword registers better suited for register allocation by compilers.

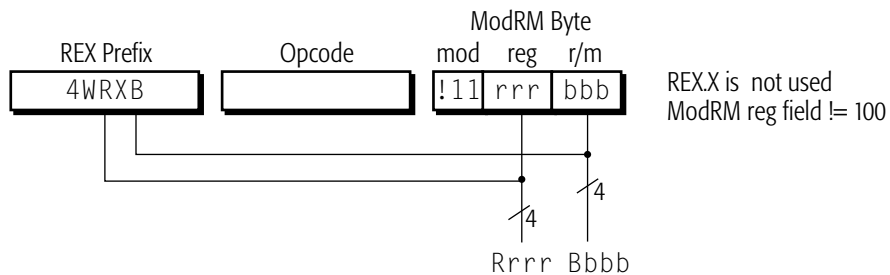
Special Encodings for Registers. Readers who need to know the details of instruction encodings should be aware that certain combinations of the ModRM and SIB fields have special meaning for register encodings. For some of these combinations, the instruction fields expanded by the REX prefix are not decoded (treated as don't cares), thereby creating aliases of these encodings in the extended registers. Table 1-12 on page 19 describes how each of these cases behaves.

Implications for INC and DEC Instructions. The REX prefix values are taken from the 16 single-byte INC and DEC instructions, one for each of the eight GPRs. Therefore, these single-byte opcodes for INC and DEC are not available in 64-bit mode, although they are available in legacy and compatibility modes. The functionality of these INC and DEC instructions is still available in 64-bit mode, however, using the ModRM forms of those instructions (opcodes FF /0 and FF /1).

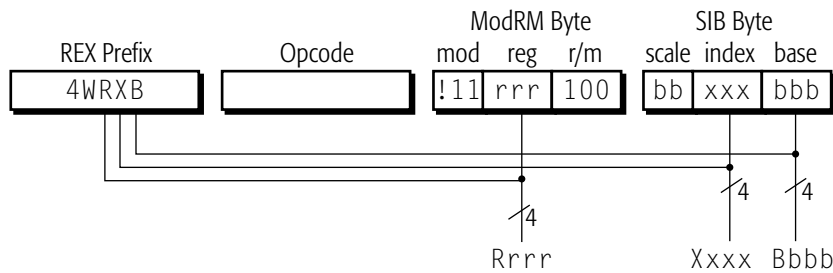
Case 1: Register-Register Addressing (No Memory Operand)



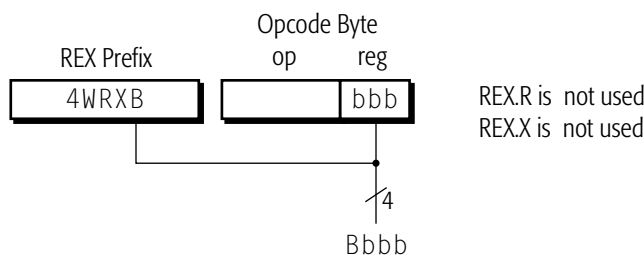
Case 2: Memory Addressing Without an SIB Byte



Case 3: Memory Addressing With an SIB Byte



Case 4: Register Operand Coded in Opcode Byte



513-302.eps

Figure 1-3. Encoding Examples of REX-Prefix R, X, and B Bits

Table 1-12. Special REX Encodings for Registers

| ModRM and SIB Encodings ² | Meaning in Legacy and Compatibility Modes | Implications in Legacy and Compatibility Modes | Additional REX Implications |
|---|---|--|---|
| ModRM Byte: <ul style="list-style-type: none"> mod \neq 11 r/m¹ = x100 (ESP) | SIB byte is present. | SIB byte is required for ESP-based addressing. | REX prefix adds a fourth bit (B), which is not decoded (don't care). Therefore, the SIB byte is also required for R12-based addressing. |
| ModRM Byte: <ul style="list-style-type: none"> mod = 00 r/m¹ = x101 (EBP) | Base register is not used. | Using EBP without a displacement must be done via mod = 01 with a displacement of 0 (with or without an index register). | REX prefix adds a fourth bit (B), which is not decoded (don't care). Therefore, using RBP or R13 without a displacement must be done via mod = 01 with a displacement of 0 (with or without an index register). |
| SIB Byte: <ul style="list-style-type: none"> index¹ = 0100 (ESP) | Index register is not used. | ESP cannot be used as an index register. | REX prefix adds a fourth bit (X), which is decoded. Therefore, there are no additional implications. The expanded index field is used to distinguish RSP from R12, allowing R12 to be used as an index. |
| Note: <ol style="list-style-type: none"> The REX-prefix bit is shown in the fourth (most-significant) bit position of the encodings for the ModRM r/m, SIB index, and SIB base fields. The lower-case "x" for ModRM r/m (rather than the upper-case "B" shown in Figure 1-3 on page 18) indicates that the REX-prefix bit is not decoded (don't care). For a description of the ModRM and SIB bytes, see "ModRM and SIB Bytes" on page 20. | | | |

1.3 Opcode

Each instruction has a unique opcode, although assemblers can support multiple mnemonics for a single instruction opcode. The opcode specifies the operation that the instruction performs and, in certain cases, the kinds of operands it uses. An opcode consists of one or two bytes, but certain 128-bit media instructions also use a prefix byte in a special way to modify the opcode. The 3-bit *reg* field of the ModRM byte ("ModRM and SIB Bytes" on page 20) is also used in certain instructions

either for three additional opcode bits or for a register specification.

128-Bit and 64-Bit Media Instruction Opcodes. Many 128-bit and 64-bit media instructions include a 66h, F2h, or F3h prefix byte in a special way to modify the opcode. These same byte values can be used in certain general-purpose and x87 instructions to modify operand size (66h) or repeat the operation (F2h, F3h). In 128-bit and 64-bit media instructions, however, such prefix bytes modify the opcode. If a 128-bit or 64-bit media instruction uses one of these three prefixes, and also includes any other prefix in the 66h, F2h, and F3h group, the result is unpredictable.

All opcodes for 64-bit media instructions begin with a 0Fh byte. In the case of 64-bit floating-point (3DNow!) instructions, the 0Fh byte is followed by a second 0Fh opcode byte. A third opcode byte occupies the same position at the end of a 3DNow! instruction as would an immediate byte. The value of the immediate byte is shown as the third opcode byte-value in the syntax for each instruction in “64-Bit Media Instruction Reference” in Volume 5. The format is:

0Fh 0Fh ModRM [SIB] [displacement] 3DNow!_third_opcode_byte

For details on opcode encoding, see Appendix A, “Opcode and Operand Encodings.”

1.4 ModRM and SIB Bytes

The ModRM byte is used in certain instruction encodings to:

- Define a register reference.
- Define a memory reference.
- Provide additional opcode bits with which to define the instruction’s function.

ModRM bytes have three fields—*mod*, *reg*, and *r/m*. The *reg* field provides additional opcode bits with which to define the instruction’s function. The *mod* and *r/m* fields are used together with each other and, in 64-bit mode, with the REX.R and REX.B bits of the REX prefix (page 14), to specify the location of an instruction’s operands and certain of the possible addressing modes (specifically, the non-complex modes).

Figure 1-4 shows the format of a ModRM byte.

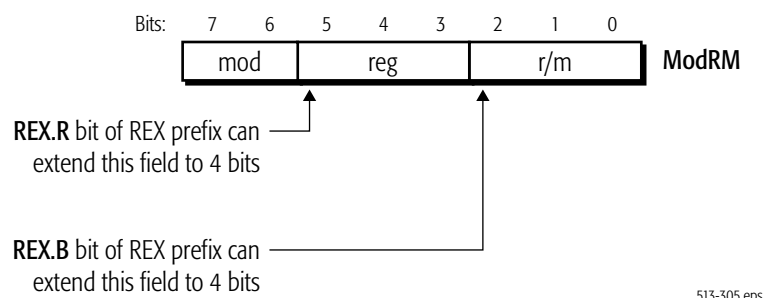


Figure 1-4. ModRM-Byte Format

In some instructions, the ModRM byte is followed by an SIB byte, which defines memory addressing for the complex-addressing modes described in “Effective Addresses” in Volume 1. The SIB byte has three fields—*scale*, *index*, and *base*—that define the scale factor, index-register number, and base-register number for 32-bit and 64-bit complex addressing modes. In 64-bit mode, the REX.B and REX.X bits extend the encoding of the SIB byte’s *base* and *index* fields.

Figure 1-5 shows the format of an SIB byte.

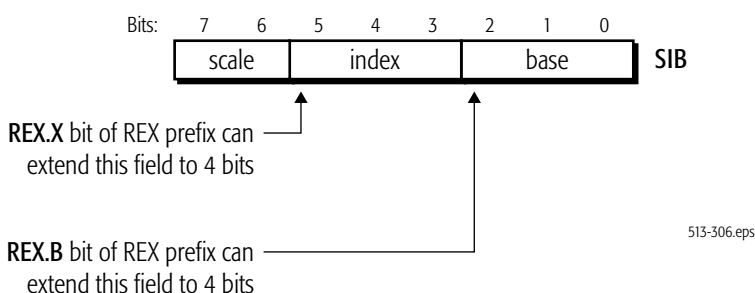


Figure 1-5. SIB-Byte Format

The encodings of ModRM and SIB bytes not only define memory-addressing modes, but they also specify operand registers. The encodings do this by using 3-bit fields in the ModRM and SIB bytes, depending on the format:

- **ModRM:** the *reg* and *r/m* fields of the ModRM byte. (Case 1 in Figure 1-3 on page 18 shows an example of this).

- *ModRM with SIB*: the *reg* field of the ModRM byte and the *base* and *index* fields of the SIB byte. (Case 3 in Figure 1-3 on page 18 shows an example of this).
- *Instructions without ModRM*: the *reg* field of the opcode. (Case 4 in Figure 1-3 on page 18 shows an example of this).

In 64-bit mode, the bits needed to extend each field for accessing the additional registers are provided by the REX prefixes, as shown in Figure 1-4 and Figure 1-5.

For details on opcode encoding, see Appendix A, “Opcode and Operand Encodings.”

1.5 Displacement Bytes

A *displacement* (also called an *offset*) is a signed value that is added to the base of a code segment (absolute addressing) or to an instruction pointer (relative addressing), depending on the addressing mode. In general, the size of a displacement is 1, 2, or 4 bytes. If an addressing mode requires a displacement, the bytes (1, 2, or 4) for the displacement follow the opcode, ModRM, or SIB byte (whichever comes last) in the instruction encoding.

In 64-bit mode, the same ModRM and SIB encodings are used to specify displacement sizes as those used in legacy and compatibility modes. However, the displacement is sign-extended to 64 bits during effective-address calculations. Also, in 64-bit mode, support is provided for some 64-bit displacement and immediate forms of the MOV instruction. See “Immediate Operand Size” in Volume 1 for more information on this.

1.6 Immediate Bytes

An *immediate* is a value—typically an operand value—encoded directly into the instruction. Depending on the opcode and the operating mode, the size of an immediate operand can be 1, 2, or 4 bytes. Immediate operands in 64-bit mode are limited to these same sizes.

If an instruction takes an immediate operand, the bytes (1, 2, or 4) for the immediate follow the opcode, ModRM, SIB, or displacement bytes (whichever come last) in the instruction

encoding. Some 128-bit media instructions use the immediate byte as a condition code.

1.7 RIP-Relative Addressing

In 64-bit mode, addressing relative to the contents of the 64-bit instruction pointer (program counter)—called RIP-relative addressing or PC-relative addressing—is implemented for certain instructions. In such cases, the effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

In the legacy x86 architecture, addressing relative to the instruction pointer is available only in control-transfer instructions. In the 64-bit mode, any instruction that uses ModRM addressing can use RIP-relative addressing. This feature is particularly useful for addressing data in position-independent code and for code that addresses global data.

Without RIP-relative addressing, ModRM instructions address memory relative to zero. With RIP-relative addressing, ModRM instructions can address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ± 2 Gbytes from the RIP.

Programs usually have many references to data, especially global data, that are not register-based. To load such a program, the loader typically selects a location for the program in memory and then adjusts program references to global data based on the load location. RIP-relative addressing of data makes this adjustment unnecessary.

1.7.1 Encoding

Table 1-13 shows the ModRM and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-only addressing exist in the current ModRM and SIB encodings. There is one ModRM encoding with several SIB encodings. RIP-relative addressing is encoded using one of the redundant forms. In 64-bit mode, the ModRM *Disp32* (32-bit displacement) encoding is redefined to be $RIP + Disp32$ rather than displacement-only.

Table 1-13. Encoding for RIP-Relative Addressing

| ModRM and SIB Encodings | Meaning in Legacy and Compatibility Modes | Meaning in 64-bit Mode | Additional 64-bit Implications |
|--|---|------------------------|---|
| ModRM Byte: <ul style="list-style-type: none"> mod = 00 r/m = 101 (none) | Disp32 | RIP + Disp32 | Zero-based (normal) displacement addressing must use SIB form (see next row). |
| SIB Byte: <ul style="list-style-type: none"> base = 101 (none) index = 100 (none) scale = 0, 1, 2, 4 | If mod = 00, Disp32 | Same as Legacy | None |

1.7.2 REX Prefix and RIP-Relative Addressing

ModRM encoding for RIP-relative addressing does not depend on a REX prefix. In particular, the *r/m* encoding of 101, used to select RIP-relative addressing, is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, *r/m* = 101) with mod = 00 still results in RIP-relative addressing.

The four-bit *r/m* field of ModRM is not fully decoded. Therefore, in order to address R13 with no displacement, software must encode it as R13 + 0 using a one-byte displacement of zero.

1.7.3 Address-Size Prefix and RIP-Relative Addressing

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. Conversely, use of the address-size prefix (“Address-Size Override Prefix” on page 6) does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits, like any other addressing mode.

2 Instruction Overview

2.1 Instruction Subsets

For easier reference, the instruction descriptions are divided into five instruction subsets. The following sections describe the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by all instructions in the x86-64 architecture:

- *Chapter 3, “General-Purpose Instruction Reference”*—The general-purpose instructions are used in basic software execution. Most of these load, store, or operate on data in the general-purpose registers (GPRs), in memory, or in both. Other instructions are used to alter sequential program flow by branching to other locations within the program or to entirely different programs.
- *Chapter 4, “System Instruction Reference”*—The system instructions establish the processor operating mode, access processor resources, handle program and system errors, and manage memory.
- *“128-Bit Media Instruction Reference” in Volume 4*—The 128-bit media instructions load, store, or operate on data located in the 128-bit XMM registers. These instructions define both vector and scalar operations on floating-point and integer data types. They include the SSE and SSE2 instructions that operate on the XMM registers. Some of these instructions convert source operands in XMM registers to destination operands in GPR, MMX™, or x87 registers or otherwise affect XMM state.
- *“64-Bit Media Instruction Reference” in Volume 5*—The 64-bit media instructions load, store, or operate on data located in the 64-bit MMX registers. These instructions define both vector and scalar operations on integer and floating-point data types. They include the legacy MMX instructions, the 3DNow!™ instructions, and the AMD extensions to the MMX and 3DNow! instruction sets. Some of these instructions convert source operands in MMX registers to destination operands in GPR, XMM, or x87 registers or otherwise affect MMX state.

- “x87 Floating-Point Instruction Reference” in Volume 5—The x87 instructions are used in legacy floating-point applications. Most of these instructions load, store, or operate on data located in the x87 ST(0)–ST(7) stack registers (the FPR0–FPR7 physical registers). The remaining instructions within this category are used to manage the x87 floating-point environment.

The description of each instruction covers its behavior in all operating modes, including legacy mode (real, virtual-8086, and protected modes) and long mode (compatibility and 64-bit modes). Details of certain kinds of complex behavior—such as control-flow changes in CALL, INT, or FXSAVE instructions—have cross-references in the instruction-detail pages to detailed descriptions in volumes 1 and 2.

Two instructions—CMPSD and MOVSD—use the same mnemonic for different instructions. Assemblers can distinguish them on the basis of the number and type of operands with which they are used.

2.2 Reference-Page Format

Figure 2-1 shows the format of an instruction-detail page. The instruction mnemonic is shown in bold at the top-left, along with its name. In this example, **POPFD** is the mnemonic and *POP to EFLAGS Doubleword* is the name. Next, there is a general description of the instruction’s operation. Many descriptions have cross-references to more detail in other parts of the manual.

Beneath the general description, the mnemonic is shown again, together with the related opcode(s) and a description summary. Related instructions are listed below this, followed by a table showing the flags that the instruction can affect. Finally, each instruction has a summary of the possible exceptions that can occur when executing the instruction. The columns labeled “Real” and “Virtual-8086” apply only to execution in legacy mode. The column labeled “Protected” applies both to legacy mode and long mode, because long mode is a superset of legacy protected mode.

The 128-bit and 64-bit media instructions also have diagrams illustrating the operation. A few instructions have examples or pseudocode describing the action.

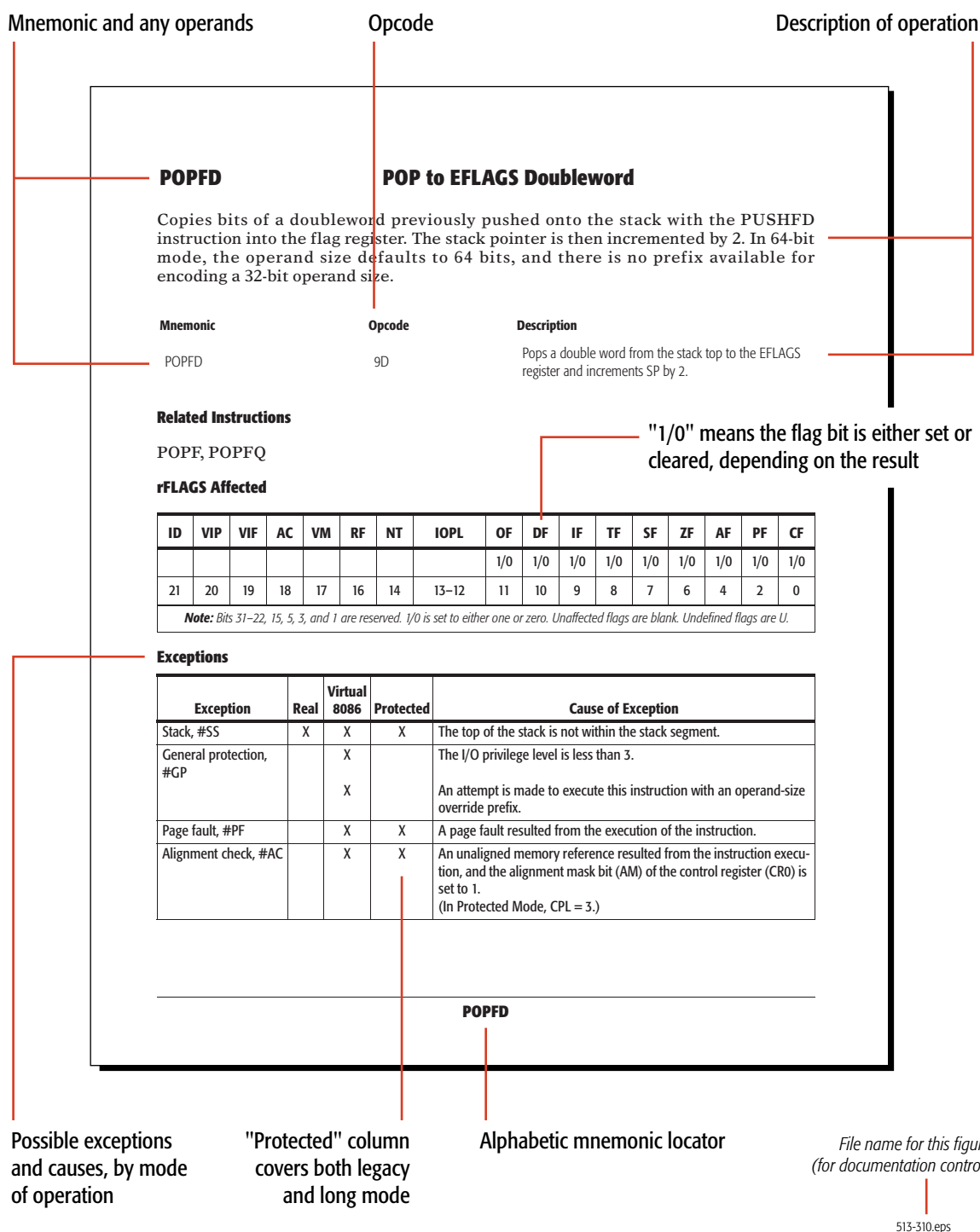


Figure 2-1. Format of Instruction-Detail Pages

2.3 Summary of Registers and Data Types

This section summarizes the registers available to software using the five instruction subsets described in “Instruction Subsets” on page 25. For details on the organization and use of these registers, see their respective chapters in volumes 1 and 2.

2.3.1 General-Purpose Instructions

Registers. The size and number of general-purpose registers (GPRs) depends on the operating mode, as do the size of the flags and instruction-pointer registers. Figure 2-2 shows the registers available in legacy and compatibility modes.

| register encoding | high 8-bit | low 8-bit | 16-bit | 32-bit |
|----------------------|---------------|--------------|--------|--------|
| 0 | AH (4) | AL | AX | EAX |
| 3 | BH (5) | BL | BX | EBX |
| 1 | CH (7) | CL | CX | ECX |
| 2 | DH (6) | DL | DX | EDX |
| 6 | SI | | SI | ESI |
| 7 | DI | | DI | EDI |
| 5 | BP | | BP | EBP |
| 4 | SP | | SP | ESP |
| 31 | 16 | 15 | 0 | |
| | | | | |
| | FLAGS | | FLAGS | EFLAGS |
| | IP | | IP | EIP |
| 31 | | | 0 | |

513-311.eps

Figure 2-2. General Registers in Legacy and Compatibility Modes

Figure 2-3 shows the registers accessible in 64-bit mode. Compared with legacy mode, registers become 64 bits wide, eight new data registers (R8–R15) are added, the low byte of all 16 GPRs is available for byte operations, and the four high-byte registers of legacy mode (AH, BH, CH, and DH) are not available. The high 32 bits of doubleword operands are zero-

extended to 64 bits, but the high bits of word and byte operands are not modified by operations in 64-bit mode. The RFLAGS register is 64 bits wide, but the high 32 bits are reserved. They can be written with anything but they read as zeros (RAZ).

| register encoding | not modified for 8-bit operands | | | | low 8-bit | 16-bit | 32-bit | 64-bit |
|----------------------|----------------------------------|-------|--------------------------------------|-------|--------------|--------|--------|--------|
| | not modified for 16-bit operands | | zero-extended for 32-bit operands | | | | | |
| | | | | | | | | |
| 0 | | | AH* | AL | AX | EAX | RAX | |
| 3 | | | BH* | BL | BX | EBX | RBX | |
| 1 | | | CH* | CL | CX | ECX | RCX | |
| 2 | | | DH* | DL | DX | EDX | RDX | |
| 6 | | | | SIL** | SI | ESI | RSI | |
| 7 | | | | DIL** | DI | EDI | RDI | |
| 5 | | | | BPL** | BP | EBP | RBP | |
| 4 | | | | SPL** | SP | ESP | RSP | |
| 8 | | | | R8B | R8W | R8D | R8 | |
| 9 | | | | R9B | R9W | R9D | R9 | |
| 10 | | | | R10B | R10W | R10D | R10 | |
| 11 | | | | R11B | R11W | R11D | R11 | |
| 12 | | | | R12B | R12W | R12D | R12 | |
| 13 | | | | R13B | R13W | R13D | R13 | |
| 14 | | | | R14B | R14W | R14D | R14 | |
| 15 | | | | R15B | R15W | R15D | R15 | |
| | 63 | 32 31 | 16 15 | 8 7 0 | | | | |

| | |
|---|--|
| 0 | |
| | |

63
32 31
0

RFLAGS
RIP

* Not addressable when a REX prefix is used.

** Only addressable when a REX prefix is used.

513-309.eps

Figure 2-3. General Registers in 64-Bit Mode

For most instructions running in 64-bit mode, access to the extended GPRs requires a REX instruction prefix (page 14).

Figure 2-4 shows the segment registers which, like the instruction pointer, are used by all instructions. In legacy and compatibility modes, all segments are accessible. In 64-bit mode, which uses the flat (non-segmented) memory model, only the CS, FS, and GS segments are recognized, whereas the contents of the DS, ES, and SS segment registers are ignored (the base for each of these segments is assumed to be zero, and neither their segment limit nor attributes are checked). For details, see “Segmented Virtual Memory” in Volume 2.

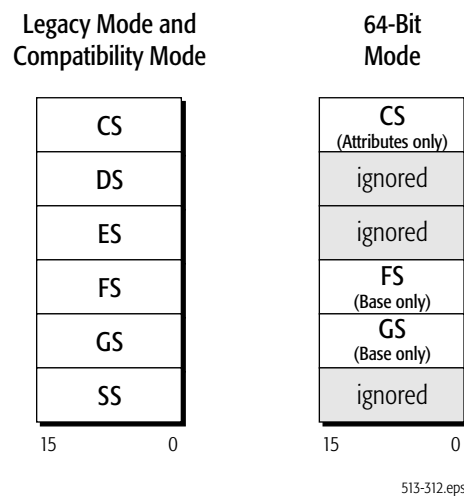


Figure 2-4. Segment Registers

Data Types. Figure 2-2 shows the general-purpose data types. They are all scalar, integer data types. The 64-bit (quadword) data types are only available in 64-bit mode, and for most instructions they require a REX instruction prefix.

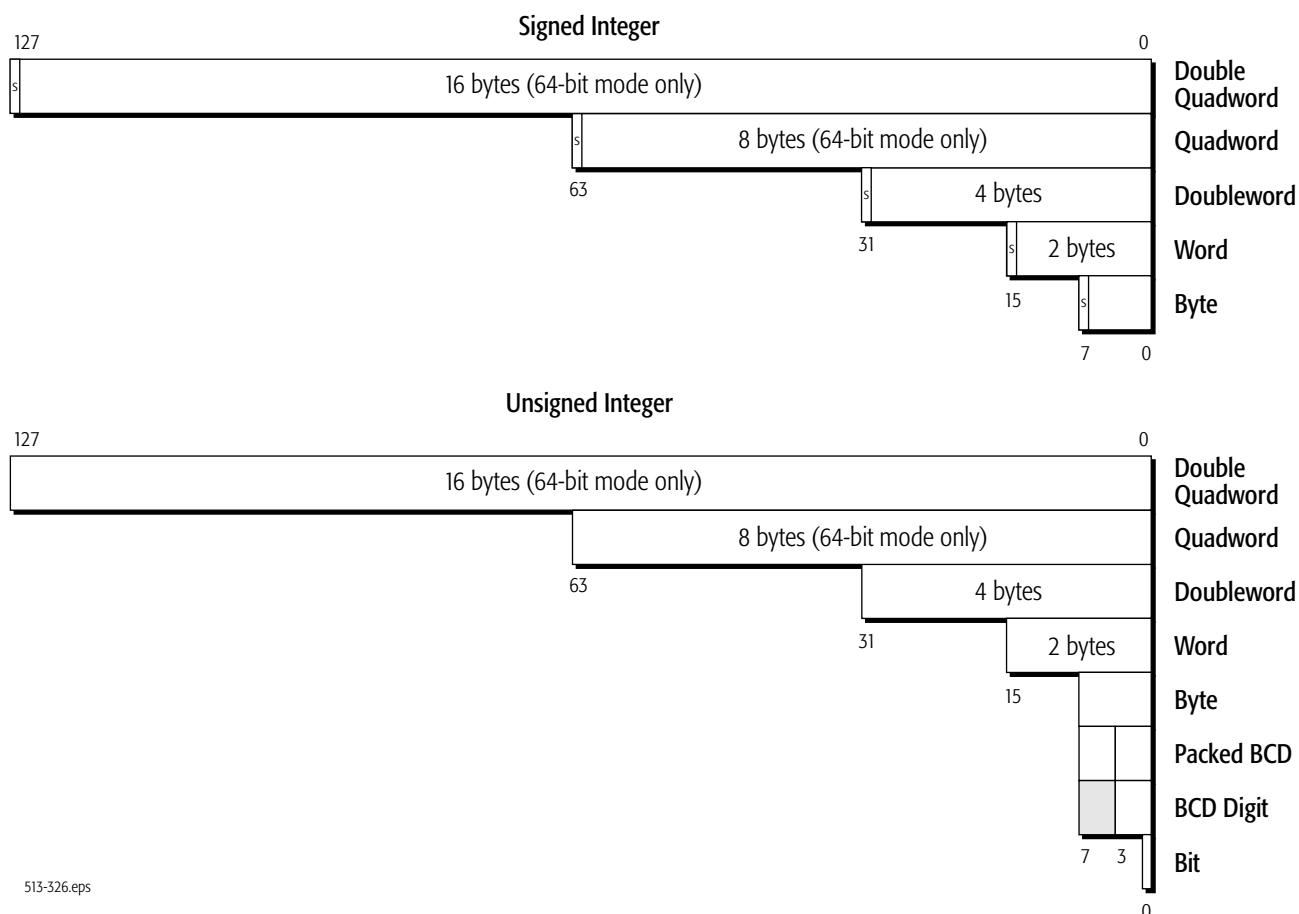
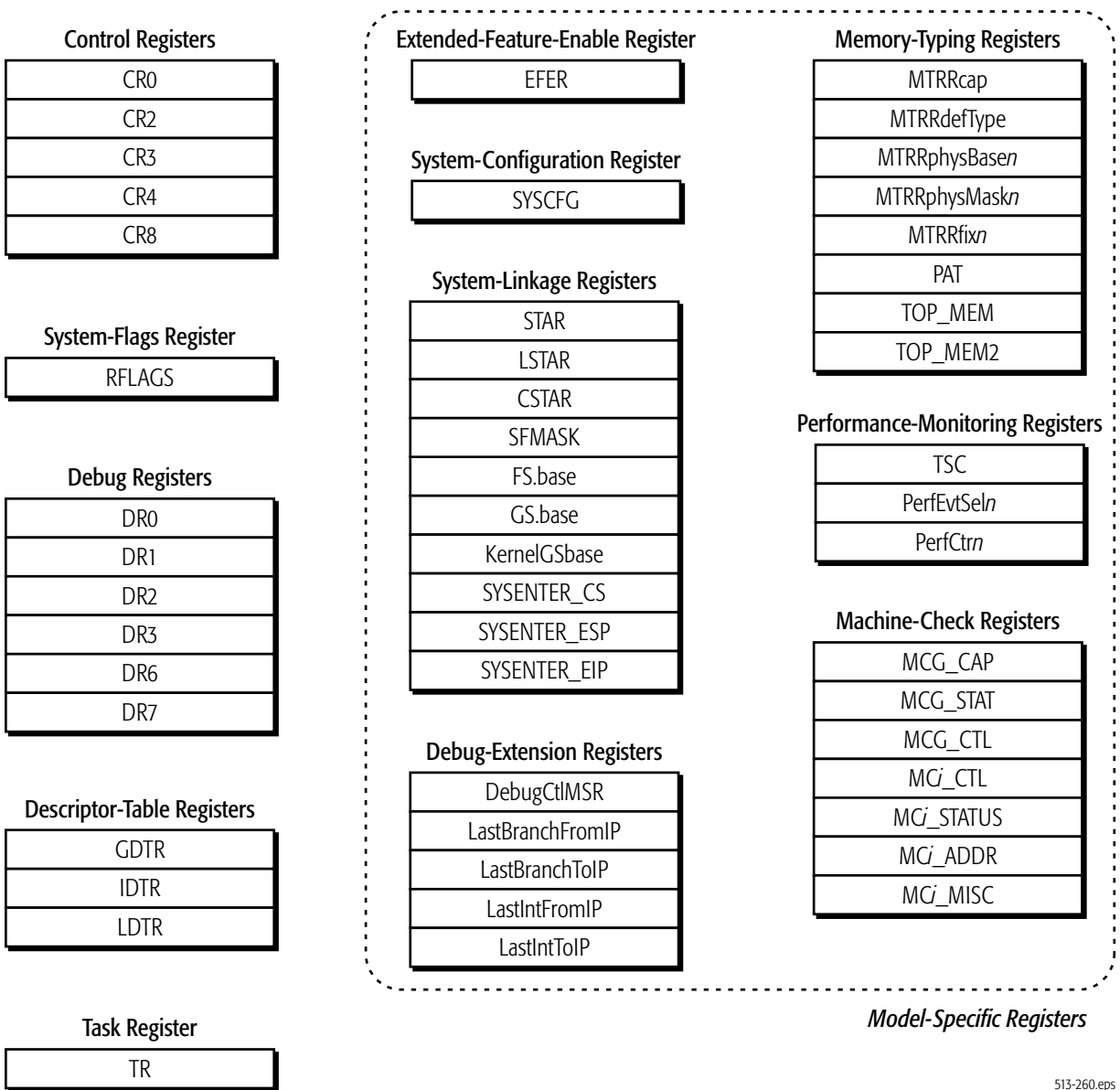


Figure 2-5. General-Purpose Data Types

2.3.2 System Instructions

Registers. The system instructions use several specialized registers shown in Figure 2-6. System software uses these registers to, among other things, manage the processor's operating environment, define system resource characteristics, and monitor software execution. With the exception of the RFLAGS register, system registers can be read and written only from privileged software.

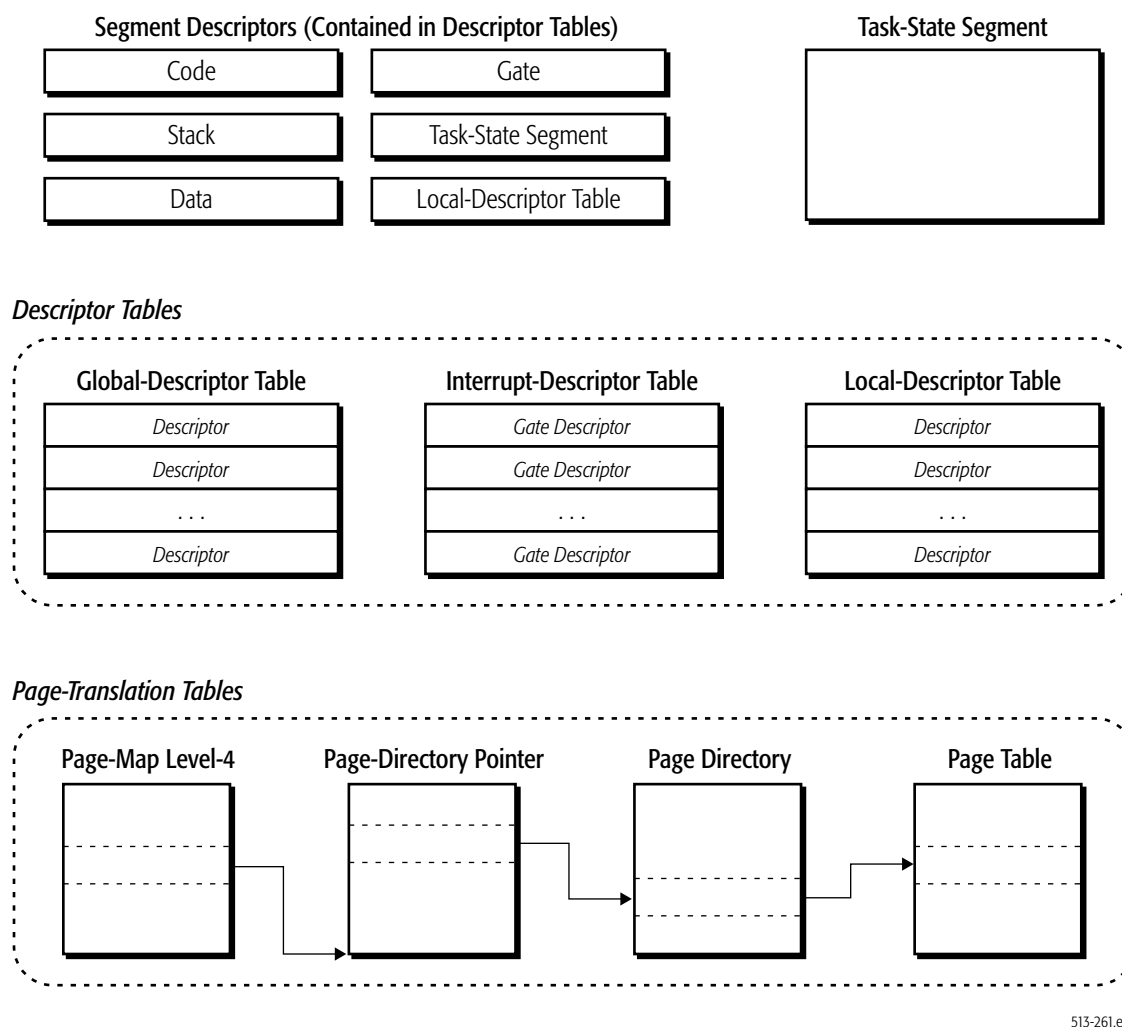
All system registers are 64 bits wide, except for the descriptor-table registers and the task register, which include 64-bit base-address fields and other fields.



513-260.eps

Figure 2-6. System Registers

Data Structures. Figure 2-7 shows the system data structures. These are created and maintained by system software for use in protected mode. A processor running in protected mode uses these data structures to manage memory and protection, and to store program-state information when an interrupt or task switch occurs.

**Figure 2-7. System Data Structures**

2.3.3 128-Bit Media Instructions

Registers. The 128-bit media instructions use the 128-bit XMM registers. The number of available XMM data registers depends on the operating mode, as shown in Figure 2-8. In legacy and compatibility modes, the eight legacy XMM data registers (XMM0–XMM7) are available. In 64-bit mode, eight additional XMM data registers (XMM8–XMM15) are available when a REX instruction prefix is used.

The MXCSR register contains floating-point and other control and status flags used by the 128-bit media instructions. Some 128-bit media instructions also use the GPR (Figure 2-2 and

Figure 2-3) and the MMX registers (Figure 2-10) or set or clear flags in the rFLAGS register (see Figure 2-2 and Figure 2-3).

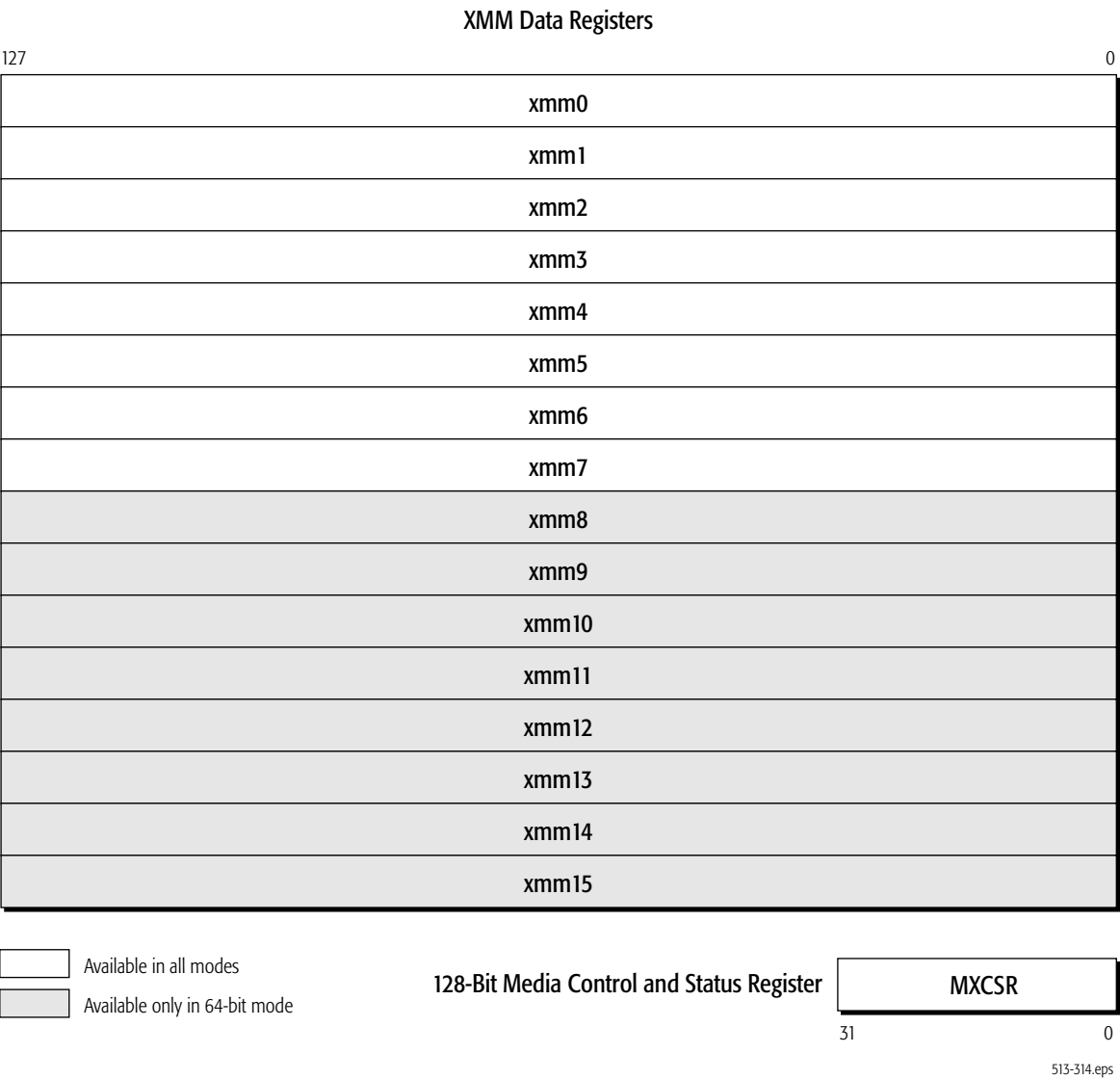


Figure 2-8. 128-Bit Media Registers

Data Types. Figure 2-9 shows the 128-bit media data types. They include floating-point and integer vectors and floating-point scalars. The floating-point data types include IEEE-754 single precision and double precision types.

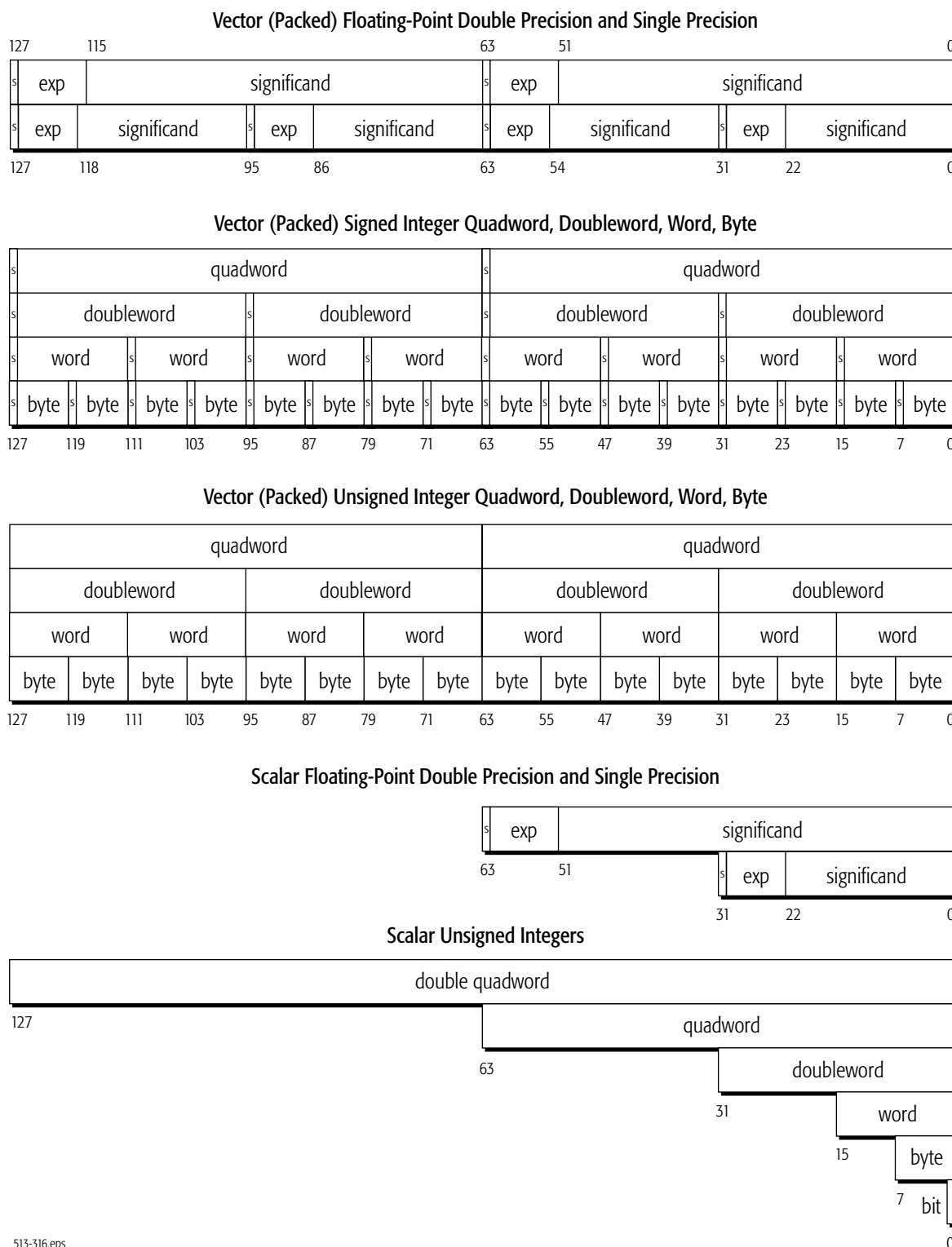
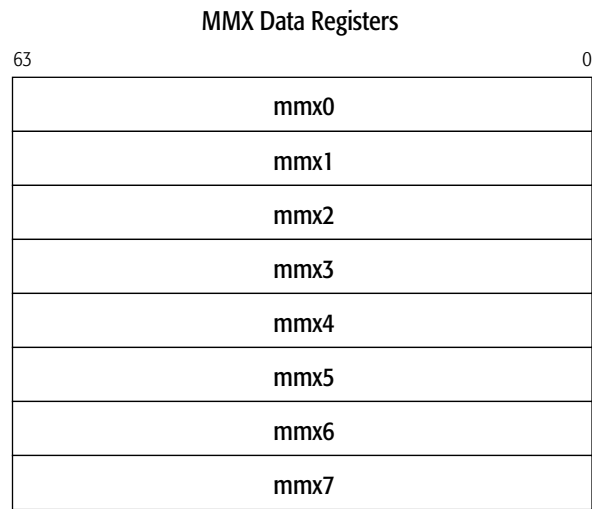


Figure 2-9. 128-Bit Media Data Types

2.3.4 64-Bit Media Instructions

Registers. The 64-bit media instructions use the eight 64-bit MMX registers, as shown in Figure 2-10. These registers are mapped onto the x87 floating-point registers, and 64-bit media instructions write the x87 tag word in a way that prevents an x87 instruction from using MMX data.

Some 64-bit media instructions also use the GPR (Figure 2-2 and Figure 2-3) and the XMM registers (Figure 2-8).



513-327eps

Figure 2-10. 64-Bit Media Registers

Data Types. Figure 2-11 shows the 64-bit media data types. They include floating-point and integer vectors and integer scalars. The floating-point data type, used by 3DNow! instructions, consists of a packed vector or two IEEE-754 32-bit single-precision data types. Unlike other kinds of floating-point instructions, however, the 3DNow! instructions do not generate floating-point exceptions. For this reason, there is no register for reporting or controlling the status of exceptions in the 64-bit-media instruction subset.

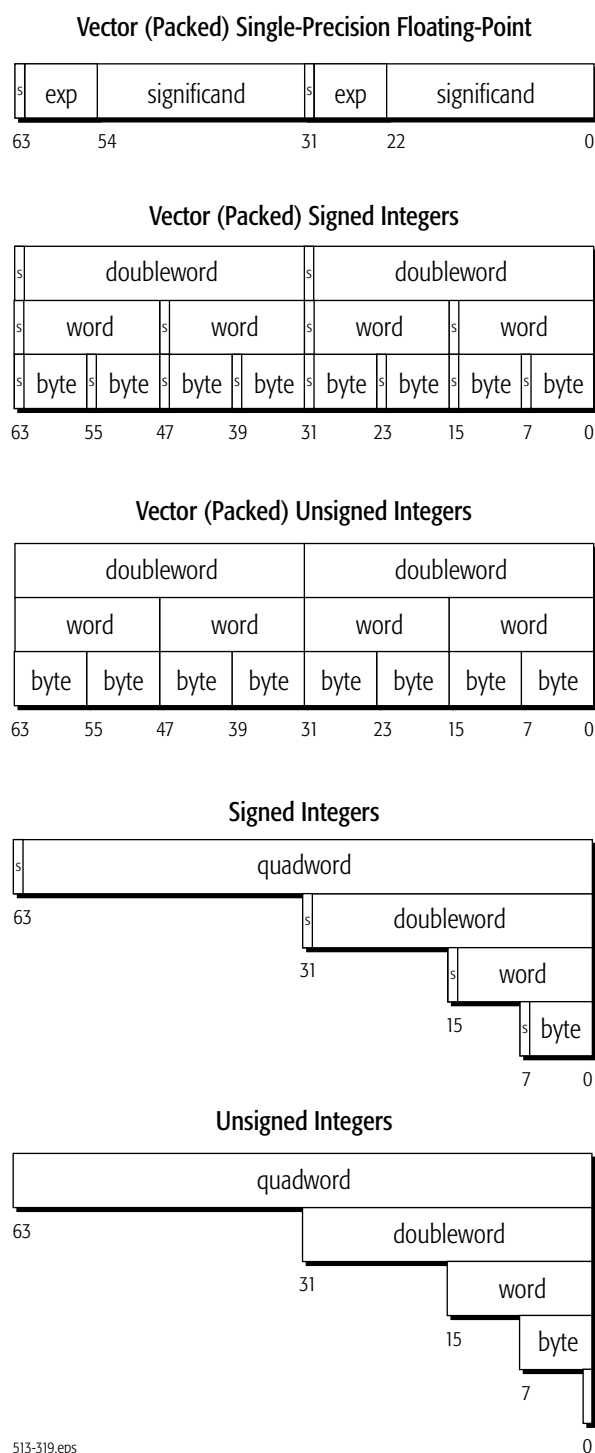


Figure 2-11. 64-Bit Media Data Types

2.3.5 x87 Floating-Point Instructions

Registers. The x87 floating-point instructions use the x87 registers shown in Figure 2-12. There are eight 80-bit data registers, three 16-bit registers that hold the x87 control word, status word, and tag word, and three registers (last instruction pointer, last opcode, last data pointer) that hold information about the last x87 operation.

The physical data registers are named FPR0–FPR7, although x87 software references these registers as a stack of registers, named ST(0)–ST(7). The x87 instructions store operands only in their own 80-bit floating-point registers or in memory. They do not access the GPR, XMM, or MMX registers.

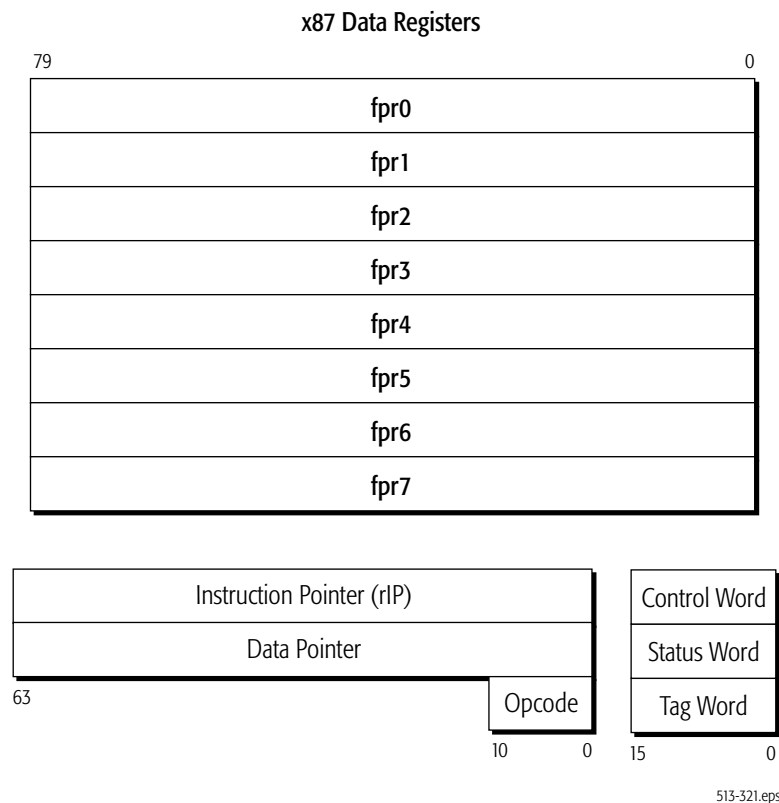
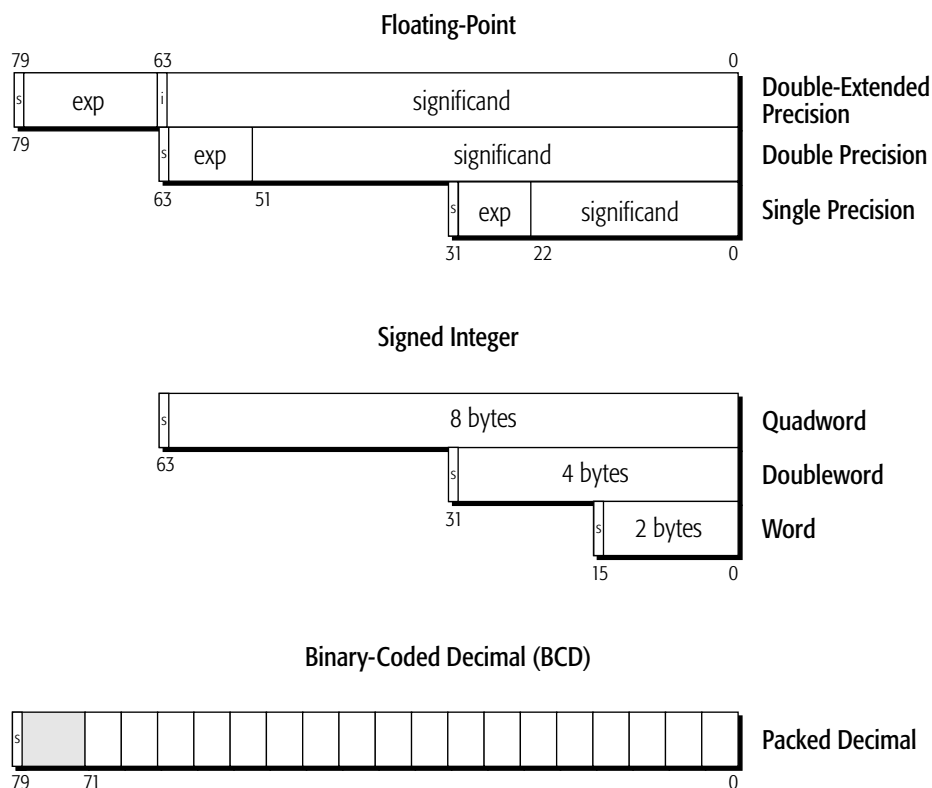


Figure 2-12. x87 Registers

Data Types. Figure 2-13 shows all x87 data types. They include three floating-point formats (80-bit double-extended precision, 64-bit double precision, and 32-bit single precision), three signed-integer formats (quadword, doubleword, and word), and an 80-bit packed binary-coded decimal (BCD) format.



513-317eps

Figure 2-13. x87 Data Types

2.4 Summary of Exceptions

Table 2-1 lists all exceptions that are possible in legacy protected mode and in long mode. The table shows the interrupt-vector numbers, names, mnemonics, source, and possible causes. Exceptions that apply to specific instructions are documented with each instruction in the instruction-detail pages that follow.

Table 2-1. Interrupt-Vector Source and Cause

| Vector | Interrupt (Exception) | Mnemonic | Source | Cause |
|--------|------------------------|----------|----------|--|
| 0 | Divide-By-Zero-Error | #DE | Software | DIV, IDIV instructions |
| 1 | Debug | #DB | Internal | Instruction accesses and data accesses |
| 2 | Non-Maskable-Interrupt | — | External | External NMI signal |

Table 2-1. Interrupt-Vector Source and Cause (continued)

| Vector | Interrupt (Exception) | Mnemonic | Source | Cause |
|--------|----------------------------------|----------|----------------------|---|
| 3 | Breakpoint | #BP | Software | INT3 instruction |
| 4 | Overflow | #OF | Software | INTO instruction |
| 5 | Bound-Range | #BR | Software | BOUND instruction |
| 6 | Invalid-Opcode | #UD | Internal | Invalid instructions |
| 7 | Device-Not-Available | #NM | Internal | x87 instructions |
| 8 | Double-Fault | #DF | Internal | Interrupt during an interrupt |
| 9 | Coprocessor-Segment-Overrun | — | External | Unsupported (reserved) |
| 10 | Invalid-TSS | #TS | Internal | Task-state segment access and task switch |
| 11 | Segment-Not-Present | #NP | Internal | Segment access through a descriptor |
| 12 | Stack | #SS | Internal | SS register loads and stack references |
| 13 | General-Protection | #GP | Internal | Memory accesses and protection checks |
| 14 | Page-Fault | #PF | Internal | Memory accesses when paging enabled |
| 15 | Reserved | — | | |
| 16 | Floating-Point Exception-Pending | #MF | Software | x87 floating-point and 64-bit media floating-point instructions |
| 17 | Alignment-Check | #AC | Internal | Memory accesses |
| 18 | Machine-Check | #MC | Internal External | Model specific |
| 19 | SIMD Floating-Point | #XF | Internal | 128-bit media floating-point instructions |
| 20–31 | Reserved (Internal and External) | — | | |
| 32–255 | External Interrupts (Maskable) | — | External | External interrupt signalling |
| 0–255 | Software Interrupts | — | Software | INT n instruction |

2.5 Notation

2.5.1 Mnemonic Syntax

Each instruction has a syntax that includes the mnemonic and any operands that the instruction can take. Figure 2-14 shows an example of a syntax in which the instruction takes two operands. In most instructions that take two operands, the first (left-most) operand is both a source operand (the first source

operand) and the destination operand. The second (right-most) operand serves only as a source, not a destination.

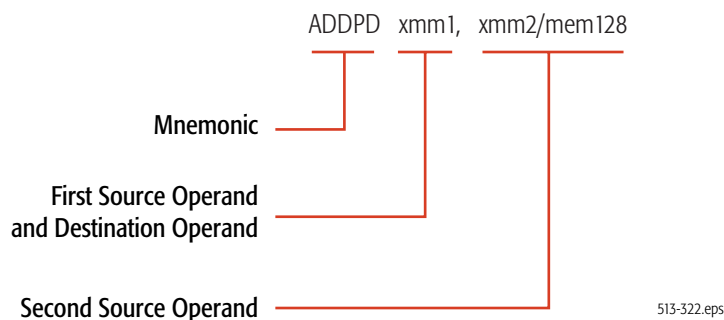


Figure 2-14. Syntax for Typical Two-Operand Instruction

The following notation is used to denote the size and type of source and destination operands:

- *cReg*—Control register.
- *dReg*—Debug register.
- *imm8*—Byte (8-bit) immediate.
- *imm16*—Word (16-bit) immediate.
- *imm16/32*—Word (16-bit) or doubleword (32-bit) immediate.
- *imm32*—Doubleword (32-bit) immediate.
- *imm32/64*—Doubleword (32-bit) or quadword (64-bit) immediate.
- *imm64*—Quadword (64-bit) immediate.
- *mem*—An operand of unspecified size in memory.
- *mem8*—Byte (8-bit) operand in memory.
- *mem16*—Word (16-bit) operand in memory.
- *mem16/32*—Word (16-bit) or doubleword (32-bit) operand in memory.
- *mem32*—Doubleword (32-bit) operand in memory.
- *mem32/48*—Doubleword (32-bit) or 48-bit operand in memory.
- *mem48*—48-bit operand in memory.
- *mem64*—Quadword (64-bit) operand in memory.
- *mem128*—Double quadword (128-bit) operand in memory.

- *mem16:16*—Two sequential word (16-bit) operands in memory.
- *mem16:32*—A word (16-bit) operand followed by a doubleword (32-bit) operand in memory.
- *mem32real*—Single-precision (32-bit) floating-point operand in memory.
- *mem32int*—Doubleword (32-bit) integer operand in memory.
- *mem64real*—Double-precision (64-bit) floating-point operand in memory.
- *mem64int*—Quadword (64-bit) integer operand in memory.
- *mem80real*—Double-extended-precision (80-bit) floating-point operand in memory.
- *mem80dec*—80-bit packed BCD operand in memory, containing 18 4-bit BCD digits.
- *mem2env*—16-bit x87 control word or x87 status word.
- *mem14/28env*—14-byte or 28-byte x87 environment. The x87 environment consists of the x87 control word, x87 status word, x87 tag word, last non-control instruction pointer, last data pointer, and opcode of the last non-control instruction completed.
- *mem94/108env*—94-byte or 108-byte x87 environment and register stack.
- *mem512env*—512-byte environment for 128-bit media, 64-bit media, and x87 instructions.
- *mmx*—Quadword (64-bit) operand in an MMX register.
- *mmx1*—Quadword (64-bit) operand in an MMX register, specified as the left-most (first) operand in the instruction syntax.
- *mmx2*—Quadword (64-bit) operand in an MMX register, specified as the right-most (second) operand in the instruction syntax.
- *mmx/mem32*—Doubleword (32-bit) operand in an MMX register or memory.
- *mmx/mem64*—Quadword (64-bit) operand in an MMX register or memory.
- *mmx1/mem64*—Quadword (64-bit) operand in an MMX register or memory, specified as the left-most (first) operand in the instruction syntax.

- *mmx2/mem64*—Quadword (64-bit) operand in an MMX register or memory, specified as the right-most (second) operand in the instruction syntax.
- *moffset*—Memory offset of unspecified size.
- *moffset8*—Operand in memory located at the specified byte (8-bit) offset from the instruction pointer.
- *moffset16*—Operand in memory located at the specified word (16-bit) offset from the instruction pointer.
- *moffset32*—Operand in memory located at the specified doubleword (32-bit) offset from the instruction pointer.
- *pntr16:16*—Far pointer with 16-bit selector and 16-bit offset.
- *pntr16:32*—Far pointer with 16-bit selector and 32-bit offset.
- *reg*—Operand of unspecified size in a GPR register.
- *reg8*—Byte (8-bit) operand in a GPR register.
- *reg16*—Word (16-bit) operand in a GPR register.
- *reg16/32*—Word (16-bit) or doubleword (32-bit) operand in a GPR register.
- *reg32*—Doubleword (32-bit) operand in a GPR register.
- *reg64*—Quadword (64-bit) operand in a GPR register.
- *reg/mem8*—Byte (8-bit) operand in a GPR register or memory.
- *reg/mem16*—Word (16-bit) operand in a GPR register or memory.
- *reg/mem32*—Doubleword (32-bit) operand in a GPR register or memory.
- *reg/mem64*—Quadword (64-bit) operand in a GPR register or memory.
- *rel8off*—Relative address in the current code segment, in 8-bit offset range.
- *rel16off*—Relative address in the current code segment, for 16-bit operand size.
- *rel32off*—Relative address in the current code segment, for 32-bit operand size.
- *segReg* or *sReg*—Word (16-bit) operand in a segment register.
- *ST(0)*—x87 stack register 0.
- *ST(i)*—x87 stack register *i*, where *i* is between 0 and 7.

- *xmm*—Double quadword (128-bit) operand in an XMM register.
- *xmm1*—Double quadword (128-bit) operand in an XMM register, specified as the left-most (first) operand in the instruction syntax.
- *xmm2*—Double quadword (128-bit) operand in an XMM register, specified as the right-most (second) operand in the instruction syntax.
- *xmm/mem64*—Quadword (64-bit) operand in a 128-bit XMM register or memory.
- *xmm/mem128*—Double quadword (128-bit) operand in an XMM register or memory.
- *xmm1/mem128*—Double quadword (128-bit) operand in an XMM register or memory, specified as the left-most (first) operand in the instruction syntax.
- *xmm2/mem128*—Double quadword (128-bit) operand in an XMM register or memory, specified as the right-most (second) operand in the instruction syntax.

2.5.2 Opcode Syntax

In addition to the notation shown above in “Mnemonic Syntax” on page 40, the following notation indicates the size and type of operands in the syntax of an instruction opcode:

- */digit*—Indicates that the ModRM byte specifies only one register or memory (r/m) operand. The digit is specified by the ModRM reg field and is used as an instruction-opcode extension. Valid digit values range from 0 to 7.
- */r*—Indicates that the ModRM byte specifies both a register operand and a reg/mem (register or memory) operand.
- *cb, cw, cd, cp*—Specifies a code-offset value and possibly a new code-segment register value. The value following the opcode is either one byte (cb), two bytes (cw), four bytes (cd), or six bytes (cp).
- *ib, iw, id*—Specifies an immediate-operand value. The opcode determines whether the value is signed or unsigned. The value following the opcode, ModRM, or SIB byte is either one byte (ib), two bytes (iw), or four bytes (id). Word and doubleword values start with the low-order byte.
- *+rb, +rw, +rd, +rq*—Specifies a register value that is added to the hexadecimal byte on the left, forming a one-byte opcode. The result is an instruction that operates on the register

specified by the register code. Valid register-code values are shown in Table 2-2.

- *m64*—Specifies a quadword (64-bit) operand in memory.
- *+i*—Specifies an x87 floating-point stack operand, ST(*i*). The value is used only with x87 floating-point instructions. It is added to the hexadecimal byte on the left, forming a one-byte opcode. Valid values range from 0 to 7.

Table 2-2. +rb, +rw, +rd, and +rq Register Value

| REX.B Bit ¹ | Value | Specified Register | | | |
|-----------------------------------|-------|--------------------|------|------|-----|
| | | +rb | +rw | +rd | +rq |
| 0 or no REX Prefix | 0 | AL | AX | EAX | RAX |
| | 1 | CL | CX | ECX | RCX |
| | 2 | DL | DX | EDX | RDY |
| | 3 | BL | BX | EBX | RBX |
| | 4 | AH | SP | ESP | RSP |
| | 5 | CH | BP | EBP | RBP |
| | 6 | DH | SI | ESI | RSI |
| | 7 | BH | DI | EDI | RDI |
| 1 | 0 | R8B | R8W | R8D | R8 |
| | 1 | R9B | R9W | R9D | R9 |
| | 2 | R10B | R10W | R10D | R10 |
| | 3 | R11B | R11W | R11D | R11 |
| | 4 | R12B | R12W | R12D | R12 |
| | 5 | R13B | R13W | R13D | R13 |
| | 6 | R14B | R14W | R14D | R14 |
| | 7 | R15B | R15W | R15D | R15 |
| 1. See "REX Prefixes" on page 14. | | | | | |

2.5.3 Pseudocode Definitions

Pseudocode examples are given for the actions of several complex instructions (for example, see “CALL” on page 85). The following definitions apply to all such pseudocode examples:

```

////////////////////////////////////////////////////////////////
// Basic Definitions
////////////////////////////////////////////////////////////////

// All comments start with these double slashes.

REAL_MODE      = (cr0.pe=0)
PROTECTED_MODE = ((cr0.pe=1) && (rflags.vm=0))
VIRTUAL_MODE   = ((cr0.pe=1) && (rflags.vm=1))
LEGACY_MODE    = (efer.lma=0)
LONG_MODE      = (efer.lma=1)
64BIT_MODE     = ((efer.lma=1) && (cs.L=1)) && cs.d=0
COMPATIBILITY_MODE = (efer.lma=1) && (cs.L=0)
PAGING_ENABLED = (cr0.pg=1)
ALIGNMENT_CHECK_ENABLED = ((cr0.am=1) && (eflags.ac=1) && (cpl=3))
CPL            = the current privilege level (0-3)
OPERAND_SIZE   = 16, 32, or 64 (depending on current code and 66h/rex prefixes)
ADDRESS_SIZE   = 16, 32, or 64 (depending on current code and 67h prefixes)
STACK_SIZE     = 16, 32, or 64 (depending on current code and SS.attr.B)

old_RIP        = RIP at the start of current instruction
old_RSP        = RSP at the start of current instruction
old_RFLAGS     = RFLAGS at the start of the instruction
old_CS         = CS selector at the start of current instruction
old_DS         = DS selector at the start of current instruction
old_ES         = ES selector at the start of current instruction
old_FS         = FS selector at the start of current instruction
old_GS         = GS selector at the start of current instruction
old_SS         = SS selector at the start of current instruction

RIP            = the current RIP register
RSP            = the current RSP register
RBP            = the current RBP register
RFLAGS        = the current RFLAGS register
NEXT_RIP       = RIP at start of next instruction

CS             = the current CS descriptor, including the subfields:
                 sel base limit attr
SS             = the current SS descriptor, including the subfields:
                 sel base limit attr

SRC            = the instruction's Source operand
DEST          = the instruction's Destination operand

temp_*         // 64-bit temporary register

```

```

temp*_desc      // temporary descriptor, with subfields:
                //     if it points to a block of memory: sel base limit attr
                //     if it's a gate descriptor: sel offset segment attr

NULL = 0x0000   // null selector is all zeros

// V,Z,A,S are integer variables, assigned a value when an instruction begins
// executing (they can be assigned a different value in the middle of an
// instruction, if needed)

V = 2 if OPERAND_SIZE=16
   4 if OPERAND_SIZE=32
   8 if OPERAND_SIZE=64

Z = 2 if OPERAND_SIZE=16
   4 if OPERAND_SIZE=32
   4 if OPERAND_SIZE=64

A = 2 if ADDRESS_SIZE=16
   4 if ADDRESS_SIZE=32
   8 if ADDRESS_SIZE=64

S = 2 if STACK_SIZE=16
   4 if STACK_SIZE=32
   8 if STACK_SIZE=64

/////////////////////////////////////////////////////////////////
// Bit Range Inside a Register
/////////////////////////////////////////////////////////////////

temp_data.[X:Y]      // Bit X through Y in temp_data, with the other bits
                    // in the register masked off.

/////////////////////////////////////////////////////////////////
// Moving Data From One Register To Another
/////////////////////////////////////////////////////////////////

temp_dest.b = temp_src // 1-byte move (copies lower 8 bits of temp_src to
                        // temp_dest, preserving the upper 56 bits of temp_dest)
temp_dest.w = temp_src // 2-byte move (copies lower 16 bits of temp_src to
                        // temp_dest, preserving the upper 48 bits of temp_dest)
temp_dest.d = temp_src // 4-byte move (copies lower 32 bits of temp_src to
                        // temp_dest, and zeros out the upper 32 bits of temp_dest)
temp_dest.q = temp_src // 8-byte move (copies all 64 bits of temp_src to
                        // temp_dest)

temp_dest.v = temp_src // 2-byte move if V=2,
                        // 4-byte move if V=4,

```

```

// 8-byte move if V=8

temp_dest.z = temp_src // 2-byte move if Z=2,
// 4-byte move if Z=4

temp_dest.a = temp_src // 2-byte move if A=2,
// 4-byte move if A=4,
// 8-byte move if A=8

temp_dest.s = temp_src // 2-byte move if S=2,
// 4-byte move if S=4,
// 8-byte move if S=8

/////////////////////////////////////////////////////////////////
// Bitwise Operations
/////////////////////////////////////////////////////////////////

temp = a AND b
temp = a OR b
temp = a XOR b
temp = NOT a
temp = a SHL b
temp = a SHR b

/////////////////////////////////////////////////////////////////
// Logical Operations
/////////////////////////////////////////////////////////////////

IF (F00 && BAR)
IF (F00 || BAR)
IF (F00 = BAR)
IF (F00 != BAR)
IF (F00 > BAR)
IF (F00 < BAR)
IF (F00 >= BAR)
IF (F00 <= BAR)

/////////////////////////////////////////////////////////////////
// IF-THEN-ELSE
/////////////////////////////////////////////////////////////////

IF (F00)
...

IF (F00)
...
ELSIF (BAR)
...

```

```

ELSE
    ...

IF ((FOO && BAR) || (CONE && HEAD))
    ...

////////////////////////////////////
// Exceptions
////////////////////////////////////

EXCEPTION [#GP(0)]      // error code in parenthesis
EXCEPTION [#UD]         // if no error code

possible exception types:

#DE      // Divide-By-Zero-Error Exception (Vector 0)
#DB      // Debug Exception (Vector 1)
#BP      // INT3 Breakpoint Exception (Vector 3)
#OF      // INTO Overflow Exception (Vector 4)
#BR      // Bound-Range Exception (Vector 5)
#UD      // Invalid-Opcode Exception (Vector 6)
#NM      // Device-Not-Available Exception (Vector 7)
#DF      // Double-Fault Exception (Vector 8)
#TS      // Invalid-TSS Exception (Vector 10)
#NP      // Segment-Not-Present Exception (Vector 11)
#SS      // Stack Exception (Vector 12)
#GP      // General-Protection Exception (Vector 13)
#PF      // Page-Fault Exception (Vector 14)
#MF      // x87 Floating-Point Exception-Pending (Vector 16)
#AC      // Alignment-Check Exception (Vector 17)
#MC      // Machine-Check Exception (Vector 18)
#XF      // SIMD Floating-Point Exception (Vector 19)

#SX      // security exception
#SHUTDOWN // shutdown

////////////////////////////////////
// READ_MEM
// General memory read. This zero-extends the data to 64 bits and returns it.
////////////////////////////////////

usage:
    temp = READ_MEM.x [seg:offset]    // where x is one of {v, z, b, w, d, q}
                                      // and denotes the size of the memory read

definition:

    IF ((seg F0xFFFC) = NULL)        // GP fault for using a null segment to
                                      // reference memory

```

```

    EXCEPTION [#GP(0)]

    IF ((seg=CS) || (seg=DS) || (seg=ES) || (seg=FS) || (seg=GS))
        // CS,DS,ES,FS,GS check for segment limit or canonical
    IF ((!64BIT_MODE) && (offset is outside seg's limit))
        EXCEPTION [#GP(0)]
        // #GP fault for segment limit violation in non-64-bit mode
    IF ((64BIT_MODE) && (offset is non-canonical))
        EXCEPTION [#GP(0)]
        // #GP fault for non-canonical address in 64-bit mode
    ELSIF (seg=SS) // SS checks for segment limit or canonical
    IF ((!64BIT_MODE) && (offset is outside seg's limit))
        EXCEPTION [#SS(0)]
        // stack fault for segment limit violation in non-64-bit mode
    IF ((64BIT_MODE) && (offset is non-canonical))
        EXCEPTION [#SS(0)]
        // stack fault for non-canonical address in 64-bit mode
    ELSE // ((seg=GDT) || (seg=LDT) || (seg=IDT) || (seg=TSS))
        // GDT,LDT,IDT,TSS check for segment limit and canonical
    IF (offset > seg.limit)
        EXCEPTION [#GP(0)] // #GP fault for segment limit violation
        // in all modes
    IF ((LONG_MODE) && (offset is non-canonical))
        EXCEPTION [#GP(0)] // #GP fault for non-canonical address in long mode

    IF ((ALIGNMENT_CHECK_ENABLED) && (offset misaligned, considering its size and
alignment))
        EXCEPTION [#AC(0)]

    IF ((64_bit_mode) && ((seg=CS) || (seg=DS) || (seg=ES) || (seg=SS)))
        temp_linear = offset
    ELSE
        temp_linear = seg.base + offset

    IF ((PAGING_ENABLED) && (virtual-to-physical translation for temp_linear
results in a page-protection violation))
        EXCEPTION [#PF(error_code)] // page fault for page-protection violation
        // (U/S violation, Reserved bit violation)

    IF ((PAGING_ENABLED) && (temp_linear is on a not-present page))
        EXCEPTION [#PF(error_code)] // page fault for not-present page

    temp_data = memory [temp_linear].x // zero-extends the data to 64
        // bits, and saves it in temp_data

    RETURN (temp_data) // return the zero-extended data

```

```

////////////////////////////////////
// WRITE_MEM // General memory write
////////////////////////////////////

```

usage:

```

WRITE_MEM.x [seg:offset] = temp.x // where <X> is one of these:
                                   // {V, Z, B, W, D, Q} and denotes the
                                   // size of the memory write

```

definition:

```

IF ((seg & 0xFFFFC) = NULL) // GP fault for using a null segment to refer-
ence memory
    EXCEPTION [#GP(0)]

IF (seg isn't writable) // GP fault for writing to a read-only segment
    EXCEPTION [#GP(0)]

IF ((seg=CS) || (seg=DS) || (seg=ES) || (seg=FS) || (seg=GS))
    // CS,DS,ES,FS,GS check for segment limit or canonical
    IF ((!64BIT_MODE) && (offset is outside seg's limit))
        EXCEPTION [#GP(0)]
        // #GP fault for segment limit violation in non-64-bit mode
    IF ((64BIT_MODE) && (offset is non-canonical))
        EXCEPTION [#GP(0)]
        // #GP fault for non-canonical address in 64-bit mode
ELIF (seg=SS) // SS checks for segment limit or canonical
    IF ((!64BIT_MODE) && (offset is outside seg's limit))
        EXCEPTION [#SS(0)]
        // stack fault for segment limit violation in non-64-bit mode
    IF ((64BIT_MODE) && (offset is non-canonical))
        EXCEPTION [#SS(0)]
        // stack fault for non-canonical address in 64-bit mode
ELSE // ((seg=GDT) || (seg=LDT) || (seg=IDT) || (seg=TSS))
    // GDT,LDT,IDT,TSS check for segment limit and canonical
    IF (offset > seg.limit)
        EXCEPTION [#GP(0)]
        // #GP fault for segment limit violation in all modes
    IF ((LONG_MODE) && (offset is non-canonical))
        EXCEPTION [#GP(0)]
        // #GP fault for non-canonical address in long mode

IF ((ALIGNMENT_CHECK_ENABLED) && (offset is misaligned, considering its size
and alignment))
    EXCEPTION [#AC(0)]

IF ((64_bit_mode) && ((seg=CS) || (seg=DS) || (seg=ES) || (seg=SS))
    temp_linear = offset
ELSE
    temp_linear = seg.base + offset

```

```

IF ((PAGING_ENABLED) && (the virtual-to-physical translation for
temp_linear results in a page-protection violation))
{
    EXCEPTION [#PF(error_code)]
        // page fault for page-protection violation
        // (U/S violation, Reserved bit violation)
}

IF ((PAGING_ENABLED) && (temp_linear is on a not-present page))
    EXCEPTION [#PF(error_code)] // page fault for not-present page

memory [temp_linear].x = temp.x // write the bytes to memory

////////////////////////////////////
// PUSH // Write data to the stack
////////////////////////////////////

usage:
    PUSH.x temp // where x is one of these: {v, z, b, w, d, q} and
                // denotes the size of the push

definition:

    WRITE_MEM.x [SS:RSP.s - X] = temp.x // write to the stack
    RSP.s = RSP - X // point rsp to the data just written

////////////////////////////////////
// POP // Read data from the stack, zero-extend it to 64 bits
////////////////////////////////////

usage:
    POP.x temp // where x is one of these: {v, z, b, w, d, q} and
               // denotes the size of the pop

definition:

    temp = READ_MEM.x [SS:RSP.s] // read from the stack
    RSP.s = RSP + X // point rsp above the data just written

////////////////////////////////////
// READ_DESCRIPTOR // Read 8-byte descriptor from GDT/LDT, return the descriptor
////////////////////////////////////

usage:
    temp_descriptor = READ_DESCRIPTOR (selector, chktype)

    // cs_chk used for far call and far jump
    // clg_chk used when reading CS for far call or far jump through call gate

```



```
// ss_chk      used when reading SS
// iret_chk     used when reading CS for IRET or RETF
// intcs_chk    used when reading the CS for interrupts and exceptions
```

definition:

```
temp_offset = selector AND 0xffff8      // upper 13 bits give an offset
                                         // in the descriptor table

IF (selector.TI = 0)                    // read 8 bytes from the gdt, split it into
                                         // (base,limit,attr) if the type bits
    temp_desc = READ_MEM.q [gdt:temp_offset]
                                         // indicate a block of memory, or split it into
                                         // (segment,offset,attr) if the type bits indicate
                                         // a gate, and save the result in temp_desc
ELSE
    temp_desc = READ_MEM.q [ldt:temp_offset]
                                         // read 8 bytes from the ldt, split it into
                                         // (base,limit,attr) if the type bits
                                         // indicate a block of memory, or split it into
                                         // (segment,offset,attr) if the type
                                         // bits indicate a gate, and save the result in
                                         // temp_desc

IF (selector.rpl or temp_desc.attr.dpl is illegal for the current mode/cpl)
    EXCEPTION [#GP(selector)]

IF (temp_desc.attr.type is illegal for the current mode/chktype)
    EXCEPTION [#GP(selector)]

IF (temp_desc.attr.p=0)
    EXCEPTION [#NP(selector)]

RETURN (temp_desc)
```

```
////////////////////////////////////
// READ_IDT // Read an 8-byte descriptor from the IDT, return the descriptor
////////////////////////////////////
```

usage:

```
temp_idt_desc = READ_IDT (vector)
                // "vector" is the interrupt vector number
```

definition:

```
IF (LONG_MODE)          // long-mode idt descriptors are 16 bytes long
    temp_offset = vector*16
ELSE // (LEGACY_MODE) legacy-protected-mode idt descriptors are 8 bytes long
    temp_offset = vector*8
```

```

temp_desc = READ_MEM.q [idt:temp_offset]
                // read 8 bytes from the idt, split it into
                // (segment,offset,attr), and save it in temp_desc

IF (temp_desc.attr.dpl is illegal for the current mode/cpl)
    // exception, with error code that indicates this idt gate
    EXCEPTION [#GP(vector*8+2)]

IF (temp_desc.attr.type is illegal for the current mode)
    // exception, with error code that indicates this idt gate
    EXCEPTION [#GP(vector*8+2)]

IF (temp_desc.attr.p=0)
    EXCEPTION [#NP(vector*8+2)]
                // segment-not-present exception, with an error code that
                // indicates this idt gate

RETURN (temp_desc)

/////////////////////////////////////////////////////////////////
// READ_INNER_LEVEL_STACK_POINTER
// Read a new stack pointer (rsp or ss:esp) from the tss
/////////////////////////////////////////////////////////////////

usage:
    temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER (new_cpl, ist_index)

definition:

IF (LONG_MODE)
    IF (ist_index>0)
        // if IST is selected, read an ISTn stack pointer from the tss
        temp_RSP = READ_MEM.q [tss:ist_index*8+28]
    ELSE // (ist_index=0)
        // otherwise read an RSPn stack pointer from the tss
        temp_RSP = READ_MEM.q [tss:new_cpl*8+4]
        temp_SS_desc.sel = NULL + new_cpl
        // in long mode, changing to lower cpl sets SS.sel to
        // NULL+new_cpl
    ELSE // (LEGACY_MODE)
    {
        temp_RSP = READ_MEM.d [tss:new_cpl*4]           // read ESPn from the tss
        temp_sel = READ_MEM.d [tss:new_cpl*8+8]         // read SSn from the tss
        temp_SS_desc = READ_DESCRIPTOR (temp_sel, ss_chk)
    }

return (temp_RSP:temp_SS_desc)

```

```

////////////////////////////////////////////////////////////////
// READ_BIT_ARRAY // Read 1 bit from a bit array in memory
////////////////////////////////////////////////////////////////

```

usage:

```
temp_value = READ_BIT_ARRAY ([mem], bit_number)
```

definition:

```

temp_BYTE = READ_MEM.b [mem + (bit_number SHR 3)]
                // read the byte containing the bit

temp_BIT = temp_BYTE SHR (bit_number & 7)
                // shift the requested bit position into bit 0

return (temp_BIT & 0x01)    // return '0' or '1'

```


3 General-Purpose Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the general-purpose instructions. General-purpose instructions are used in basic software execution. Most of these instructions load, store, or operate on data located in the general-purpose registers (GPRs), in memory, or in both. The remaining instructions are used to alter the sequential flow of the program by branching to other locations within the program, or to entirely different programs. The instructions within this category do not operate on any other register set.

General-purpose instructions are supported in all hardware implementations of the x86-64 architecture, except that the following general-purpose instructions are implemented only if their associated CPUID function bit is set:

- **CMPXCHG8B**, indicated by bit 8 of CPUID standard function 1 and extended function 8000_0001h.
- **CMOVcc** (conditional moves), indicated by bit 15 of CPUID standard function 1 and extended function 8000_0001h.
- **Long Mode instructions**, indicated by bit 29 of CPUID extended function 8000_0001h.

The general-purpose instructions can be used in legacy mode or 64-bit long mode. Compilation of general-purpose programs for execution in 64-bit long mode offers three primary advantages: access to the eight extended, 64-bit general-purpose registers (for a register set consisting of GPR0–GPR15), access to the 64-bit virtual address space, and access to the RIP-relative addressing mode.

For further information about the general-purpose instructions and register resources, see:

- “General-Purpose Programming” in Volume 1.
- “Summary of Registers and Data Types” on page 28.
- “Notation” on page 40.
- “Instruction Prefixes” on page 3.
- Appendix B, “General-Purpose Instructions in 64-Bit Mode.” In particular, see “General Rules for 64-Bit Mode” on page 419.

AAA ASCII Adjust After Addition

Converts the value in the AL register to an unpacked BCD value. Use the AAA instruction after using the ADD instruction to add two unpacked BCD numbers.

If the value being converted is greater than 9 or if the AF flag is set to 1, the instruction increments the AH register and sets the CF and AF flags to 1. If the value is not greater than 9, it does not change the AH register and clears the CF and AF flags to 0. In either case, it clears bits 7–4 of the AL register, leaving the correct decimal digit in bits 3–0.

This instruction makes it possible to add ASCII numbers without having to mask off the upper nibble ‘3’.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| AAA | 37 | Create an unpacked BCD number. (Invalid in 64-bit mode.) |

Related Instructions

AAD, AAM, AAS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|-----|----|-----|
| | | | | | | | | U | | | | U | U | 1/0 | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

AAD**ASCII Adjust Before Division**

Converts two unpacked BCD digits in the AL (least significant) and AH (most significant) registers to a single binary value in the AL register using the following formula: $AL = ((10d * AH) + (AL))$. After the conversion, it clears AH to 00h.

This instruction allows a binary division on the result by an unpacked BCD value to return the correct unpacked BCD value.

Using this instruction before the DIV instruction allows the division of ASCII numbers. Before the AAD instruction is executed, the ASCII tag of 3 must be masked from the upper nibble of AH and AL.

In most modern assemblers, the AAD instruction adjusts to ASCII (base-10) values. However, by coding the instruction directly in binary, it can adjust to any base specified by the immediate byte value (*ib*) suffixed onto the D5h opcode. For example, code D508h for octal, D50Ah for decimal, and D50Ch for duodecimal (base 12).

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------------|--|
| AAD | D5 0A | Adjust two BCD digits in AL and AH. (Invalid in 64-bit mode.) |
| (None) | D5 <i>ib</i> | Adjust two BCD digits to the immediate byte base. (Invalid in 64-bit mode.) |

Related Instructions

AAA, AAM, AAS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|----|-----|----|
| | | | | | | | | U | | | | 1/0 | 1/0 | U | 1/0 | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|-------------|-------------------------|------------------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

AAM**ASCII Adjust After Multiply**

Converts the value in the AX register from binary to two unpacked BCD digits in the AH (most significant) and AL (least significant) registers. Use the AAM instruction after using the MUL instruction to multiply two unpacked BCD numbers.

In most modern assemblers, the AAM instruction adjusts to ASCII (base-10) values. However, by coding the instruction directly in binary, it can adjust to any base specified by the immediate byte value (*ib*) suffixed onto the D5h opcode. For example, code D508h for octal, D50Ah for decimal, and D50Ch for duodecimal (base 12).

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------------|---|
| AAM | D4 0A | Create a pair of unpacked BCD values in AH and AL. (Invalid in 64-bit mode.) |
| (None) | D4 <i>ib</i> | Create a pair of unpacked values to the immediate byte base. (Invalid in 64-bit mode.) |

Related Instructions

AAA, AAD, AAS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|----|-----|----|
| | | | | | | | | U | | | | 1/0 | 1/0 | U | 1/0 | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |
| Divide by zero, #DE | X | X | X | 8-bit immediate value was 0. |

AAS**ASCII Adjust After Subtraction**

Converts the binary value in the AL register to a one-digit unpacked BCD result. If the value in AL is greater than 9 or the AF flag is set to 1, the instruction decrements the value in AH and sets the CF and AF flags to 1. Otherwise, it clears the CF and AF flags and the AH register is unchanged. Finally, the instruction clears bits 7–4 of the AL register to 0.

A binary subtraction of one unpacked BCD value from another leaves a byte result in the AL register. The AAS instruction adjusts this result, leaving the correct one-digit unpacked BCD result in AL.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| AAS | 3F | Create an unpacked BCD number from the contents of the AL register. (Invalid in 64-bit mode.) |

Related Instructions

AAA, AAD, AAM

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|-----|----|-----|
| | | | | | | | | U | | | | U | U | 1/0 | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

ADC Add with Carry

Adds the the carry flag (CF), the value in a register or memory location (first operand), and an immediate value or the value in a register a memory location (second operand), and stores the result in the first operand location. The instruction cannot add two memory operands. The CF flag indicates a carry from a previous addition operation. The instruction sign-extends an immediate value to the length of the destination register or memory location.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

Use the ADC instruction after an ADD instruction as part of a multibyte or multiword addition.

The memory-operand variants of the ADC instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|---|
| ADC AL, <i>imm8</i> | 14 <i>ib</i> | Add <i>imm8</i> to AL with carry. |
| ADC AX, <i>imm16</i> | 15 <i>iw</i> | Add <i>imm16</i> to AX with carry. |
| ADC EAX, <i>imm32</i> | 15 <i>id</i> | Add <i>imm32</i> to EAX with carry. |
| ADC RAX, <i>imm32</i> | 15 <i>id</i> | Add sign-extended <i>imm32</i> to RAX with carry. |
| ADC <i>reg/mem8</i> , <i>imm8</i> | 80 /2 <i>ib</i> | Add <i>imm8</i> to byte register or memory location with carry. |
| ADC <i>reg/mem16</i> , <i>imm16</i> | 81 /2 <i>iw</i> | Add <i>imm16</i> to word register or memory location with carry. |
| ADC <i>reg/mem32</i> , <i>imm32</i> | 81 /2 <i>id</i> | Add <i>imm32</i> to double register or memory location with carry flag. |
| ADC <i>reg/mem64</i> , <i>imm32</i> | 81 /2 <i>id</i> | Add sign-extended <i>imm32</i> to quadword register or memory location with carry flag. |
| ADC <i>reg/mem16</i> , <i>imm8</i> | 83 /2 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem16</i> with sign-extension based on CF. |
| ADC <i>reg/mem32</i> , <i>imm8</i> | 83 /2 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem32</i> with sign-extension based on CF. |
| ADC <i>reg/mem64</i> , <i>imm8</i> | 83 /2 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem64</i> with sign-extension based on CF. |

| Mnemonic | Opcode | Description |
|-----------------------------|--------|--|
| ADC <i>reg/mem8, reg8</i> | 10 /r | Add contents of byte memory location or byte register to contents of byte register with carry. |
| ADC <i>reg/mem16, reg16</i> | 11 /r | Add contents of word memory location or word register to contents of word register with carry. |
| ADC <i>reg/mem32, reg32</i> | 11 /r | Add contents of double memory location or double register to contents of double register with carry. |
| ADC <i>reg/mem64, reg64</i> | 11 /r | Add contents of quadword memory location or quadword register to contents of quadword register with carry. |
| ADC <i>reg8, reg/mem8</i> | 12 /r | Add contents of byte register or memory location to byte register with carry. |
| ADC <i>reg16, reg/mem16</i> | 13 /r | Add contents of word register or memory location to word register with carry. |
| ADC <i>reg32, reg/mem32</i> | 13 /r | Add contents of doubleword register or memory location to doubleword register with sign-extended carry flag. |
| ADC <i>reg64, reg/mem64</i> | 13 /r | Add contents of quadword register or memory location to quadword register with sign-extended carry flag. |

Related Instructions

ADD, SUB

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

ADD Signed or Unsigned Add

Adds the value in a register or memory location (first operand) and an immediate value or the value in a register a memory location (second operand), and stores the result in the first operand location. The instruction cannot add two memory operands. The instruction sign-extends an immediate value to the length of the destination register or memory operand.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

The memory-operand variants of the ADD instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|--|
| ADD AL, <i>imm8</i> | 04 <i>ib</i> | Add <i>imm8</i> to AL. |
| ADD AX, <i>imm16</i> | 05 <i>iw</i> | Add <i>imm16</i> to AX. |
| ADD EAX, <i>imm32</i> | 05 <i>id</i> | Add <i>imm32</i> to EAX. |
| ADD RAX, <i>imm32</i> | 05 <i>id</i> | Add sign-extended <i>imm32</i> to RAX. |
| ADD <i>reg/mem8</i> , <i>imm8</i> | 80 /0 <i>ib</i> | Add <i>imm8</i> to <i>reg/mem8</i> . |
| ADD <i>reg/mem16</i> , <i>imm16</i> | 81 /0 <i>iw</i> | Add <i>imm16</i> to <i>reg/mem16</i> . |
| ADD <i>reg/mem32</i> , <i>imm32</i> | 81 /0 <i>id</i> | Add <i>imm32</i> to <i>reg/mem32</i> . |
| ADD <i>reg/mem64</i> , <i>imm32</i> | 81 /0 <i>id</i> | Add sign-extended <i>imm32</i> to <i>reg/mem64</i> . |
| ADD <i>reg/mem16</i> , <i>imm8</i> | 83 /0 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem8</i> . |
| ADD <i>reg/mem32</i> , <i>imm8</i> | 83 /0 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem32</i> . |
| ADD <i>reg/mem64</i> , <i>imm8</i> | 83 /0 <i>ib</i> | Add sign-extended <i>imm8</i> to <i>reg/mem64</i> . |
| ADD <i>reg/mem8</i> , <i>reg8</i> | 00 /r | Add <i>reg8</i> to <i>reg/mem8</i> . |
| ADD <i>reg/mem16</i> , <i>reg16</i> | 01 /r | Add <i>reg16</i> to <i>reg/mem16</i> . |
| ADD <i>reg/mem32</i> , <i>reg32</i> | 01 /r | Add <i>reg32</i> to <i>reg/mem32</i> . |
| ADD <i>reg/mem64</i> , <i>reg64</i> | 01 /r | Add <i>reg64</i> to <i>reg/mem64</i> . |
| ADD <i>reg8</i> , <i>reg/mem8</i> | 02 /r | Add <i>reg/mem8</i> to <i>reg8</i> . |

| Mnemonic | Opcode | Description |
|----------------------|--------|-------------------------|
| ADD reg16, reg/mem16 | 03 /r | Add reg/mem16 to reg16. |
| ADD reg32, reg/mem32 | 03 /r | Add reg/mem32 to reg32. |
| ADD reg64, reg/mem64 | 03 /r | Add reg/mem64 to reg64. |

Related Instructions

SUB, ADC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

AND Logical AND

Performs a bitwise AND operation on the value in a register or memory location (first operand) and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location. The instruction cannot AND two memory operands.

The instruction sets each bit of the result to 1 if the corresponding bit of both operands is set; otherwise, it clears the bit to 0. The following table shows the truth table for the AND operation:

| X | Y | X AND Y |
|---|---|---------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The memory-operand variants of the AND instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|--|
| AND AL, <i>imm8</i> | 24 <i>ib</i> | AND the contents of AL with an immediate 8-bit value and store the result in AL. |
| AND AX, <i>imm16</i> | 25 <i>iw</i> | AND the contents of AX with an immediate 16-bit value and store the result in AX. |
| AND EAX, <i>imm32</i> | 25 <i>id</i> | AND the contents of EAX with an immediate 32-bit value and store the result in EAX. |
| AND RAX, <i>imm32</i> | 25 <i>id</i> | AND the contents of RAX with a sign-extended immediate 32-bit value and store the result in RAX. |
| AND <i>reg/mem8</i> , <i>imm8</i> | 80 /4 <i>ib</i> | AND the contents of <i>reg/mem8</i> with <i>imm8</i> . |
| AND <i>reg/mem16</i> , <i>imm16</i> | 81 /4 <i>iw</i> | AND the contents of <i>reg/mem16</i> with <i>imm16</i> . |
| AND <i>reg/mem32</i> , <i>imm32</i> | 81 /4 <i>id</i> | AND the contents of <i>reg/mem32</i> with <i>imm32</i> . |
| AND <i>reg/mem64</i> , <i>imm32</i> | 81 /4 <i>id</i> | AND the contents of <i>reg/mem64</i> with sign-extended <i>imm32</i> . |

| Mnemonic | Opcode | Description |
|-----------------------------|-----------------|--|
| AND <i>reg/mem16, imm8</i> | 83 /4 <i>ib</i> | AND the contents of <i>reg/mem16</i> with a sign-extended 8-bit value. |
| AND <i>reg/mem32, imm8</i> | 83 /4 <i>ib</i> | AND the contents of <i>reg/mem32</i> with a sign-extended 8-bit value. |
| AND <i>reg/mem64, imm8</i> | 83 /4 <i>ib</i> | AND the contents of <i>reg/mem64</i> with a sign-extended 8-bit value. |
| AND <i>reg/mem8, reg8</i> | 20 / <i>r</i> | AND the contents of an 8-bit register or memory location with the contents of an 8-bit register. |
| AND <i>reg/mem16, reg16</i> | 21 / <i>r</i> | AND the contents of a 16-bit register or memory location with the contents of a 16-bit register. |
| AND <i>reg/mem32, reg32</i> | 21 / <i>r</i> | AND the contents of a 32-bit register or memory location with the contents of a 32-bit register. |
| AND <i>reg/mem64, reg64</i> | 21 / <i>r</i> | AND the contents of a 64-bit register or memory location with the contents of a 64-bit register. |
| AND <i>reg8, reg/mem8</i> | 22 / <i>r</i> | AND the contents of an 8-bit register with the contents of an 8-bit memory location or register. |
| AND <i>reg16, reg/mem16</i> | 23 / <i>r</i> | AND the contents of a 16-bit register with the contents of a 16-bit memory location or register. |
| AND <i>reg32, reg/mem32</i> | 23 / <i>r</i> | AND the contents of a 32-bit register with the contents of a 32-bit memory location or register. |
| AND <i>reg64, reg/mem64</i> | 23 / <i>r</i> | AND the contents of a 64-bit register with the contents of a 64-bit memory location or register. |

Related Instructions

TEST, OR, NOT, NEG

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|----|-----|----|
| | | | | | | | | 0 | | | | 1/0 | 1/0 | U | 1/0 | 0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BOUND Check Array Bounds

Checks whether an array index (first operand) is within the bounds of an array (second operand). The array index is a signed integer in the specified register. If the operand-size attribute is 16, the array operand is a memory location containing a pair of signed word-integers; if the operand-size attribute is 32, the array operand is a pair of signed doubleword-integers. The first word or doubleword specifies the lower bound of the array and the second word or doubleword specifies the upper bound.

The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within the specified bounds, the processor generates a BOUND range-exceeded exception (#BR) and saves a return instruction pointer (RIP) to the BOUND instruction.

The bounds of an array, consisting of two words or doublewords containing the lower and upper limits of the array, usually reside in a data structure just before the array itself, making the limits addressable through a constant offset from the beginning of the array. With the address of the array in a register, this practice reduces the number of bus cycles required to determine the effective address of the array bounds.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|-------------------------------------|--------|--|
| BOUND <i>reg32, mem32&mem32</i> | 62/r | Test whether a 32-bit array index is within the bounds specified by the two 32-bit values in <i>mem32&mem32</i> . (Invalid in 64-bit mode.) |
| BOUND <i>reg16, mem16&mem16</i> | 62/r | Test whether a 16-bit array index is within the bounds specified by the two 16-bit values in <i>mem16&mem16</i> . (Invalid in 64-bit mode.) |

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Bound range, #BR | X | X | X | The bound range was exceeded. |
| Invalid opcode, #UD | X | X | X | The source operand was a register. |
| | | | X | Software is executing in 64-bit mode. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BSF**Bit Scan Forward**

Searches the value in a register or a memory location (second operand) for the least-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the least-significant set bit in a destination register (first operand). The bit index is an unsigned offset from bit 0 of the searched value. If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register.

| Mnemonic | Opcode | Description |
|-----------------------------|--------|--|
| BSF <i>reg16, reg/mem16</i> | 0F BC | Bit scan forward on the contents of <i>reg16</i> . |
| BSF <i>reg32, reg/mem32</i> | 0F BC | Bit scan forward on the contents of <i>reg32</i> . |
| BSF <i>reg64, reg/mem64</i> | 0F BC | Bit scan forward on the contents of <i>reg64</i> . |

Related Instructions

BSR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | U | | | | U | 1/0 | U | U | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BSR**Bit Scan Reverse**

Searches the value in a register or a memory location (second operand) for the most-significant set bit. If a set bit is found, the instruction clears the zero flag (ZF) and stores the index of the most-significant set bit in a destination register (first operand). The bit index is an unsigned offset from bit 0 of the searched value. If the second operand contains 0, the instruction sets ZF to 1 and does not change the contents of the destination register.

| Mnemonic | Opcode | Description |
|-----------------------------|--------|--|
| BSR <i>reg16, reg/mem16</i> | 0F BD | Bit scan reverse on the contents of <i>reg/mem16</i> . |
| BSR <i>reg32, reg/mem32</i> | 0F BD | Bit scan reverse on the contents of <i>reg/mem32</i> . |
| BSR <i>reg64, reg/mem64</i> | 0F BD | Bit scan reverse on the contents of <i>reg/mem64</i> . |

Related Instructions

BSF

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | U | | | | U | 1/0 | U | U | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BSWAP Byte Swap

Reverses the byte order of the specified register. This action converts the contents of the register from little endian to big endian or vice versa. In a doubleword, bits 7–0 are exchanged with bits 31–24, and bits 15–8 are exchanged with bits 23–16. In a quadword, bits 7–0 are exchanged with bits 63–56, bits 15–8 with bits 55–48, bits 23–16 with bits 47–40, and bits 31–24 with bits 39–32. A subsequent use of the BSWAP instruction with the same operand restores the original value of the operand.

The result of applying the BSWAP instruction to a 16-bit register is undefined. To swap the bytes of a 16-bit register, use the XCHG instruction and specify the respective byte halves of the 16-bit register as the two operands. For example, to swap the bytes of AX, use XCHG AL, AH.

| Mnemonic | Opcode | Description |
|--------------------|-----------------|--|
| BSWAP <i>reg32</i> | 0F C8 <i>rd</i> | Reverse the byte order of <i>reg32</i> . |
| BSWAP <i>reg64</i> | 0F C8 <i>rd</i> | Reverse the byte order of <i>reg64</i> . |

Related Instructions

CMPXCHG, XCHG

rFLAGS Affected

None

Exceptions

None

BT**Bit Test**

Copies a bit, specified by a zero-based offset index in a register or an 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register. If the bit base is in a register, the value in the register is the bit string. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the register size) of the bit offset index to select a bit position in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. The instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ for a register offset and 0 to 63 for an immediate value offset.

When the instruction attempts to copy a bit from memory, it accesses 2, 4, or 8 bytes starting from the specified memory address for 16-bit, 32-bit, or 64-bit operand sizes, respectively, using the following formula:

Effective Address + (NumBytes_i * (BitOffset DIV NumBits_i*8))

When using this bit addressing mechanism, avoid referencing areas of memory close to address space holes, such as references to memory-mapped I/O registers. Instead, use a MOV instruction to load a register from such an address and use a register form of the BT instruction to manipulate the data.

| Mnemonic | Opcode | Description |
|----------------------------|--------------------|--|
| BT <i>reg/mem16, reg16</i> | 0F A3 | Set the carry flag to the value of the selected bit. |
| BT <i>reg/mem32, reg32</i> | 0F A3 | Set the carry flag to the value of the selected bit. |
| BT <i>reg/mem64, reg64</i> | 0F A3 | Set the carry flag to the value of the selected bit. |
| BT <i>reg/mem16, imm8</i> | 0F BA /4 <i>ib</i> | Set the carry flag to the value of the selected bit. |
| BT <i>reg/mem32, imm8</i> | 0F BA /4 <i>ib</i> | Set the carry flag to the value of the selected bit. |
| BT <i>reg/mem64, imm8</i> | 0F BA /4 <i>ib</i> | Set the carry flag to the value of the selected bit. |

Related Instructions

BTC, BTR, BTS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|-----|
| | | | | | | | | U | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BTC**Bit Test and Complement**

Copies a bit, specified by a zero-based offset index in a register or an 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then complements (toggles) the bit in the bit string. If the bit base is in a register, the value in the register is the bit string. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the register size) of the bit offset index to select a bit position in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. The instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ for a register offset and 0 to 63 for an immediate value offset.

This instruction is useful for implementing semaphores in concurrent operating systems. Such an application should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-----------------------------|---------------|---|
| <i>BTC reg/mem16, reg16</i> | 0F BB | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |
| <i>BTC reg/mem32, reg32</i> | 0F BB | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |
| <i>BTC reg/mem64, reg64</i> | 0F BB | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |
| <i>BTC reg/mem16, imm8</i> | 0F BA /7 ib | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |
| <i>BTC reg/mem32, imm8</i> | 0F BA /7 ib | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |
| <i>BTC reg/mem64, imm8</i> | 0F BA /7 ib | Set the carry flag to the value of the selected bit and complement the bit in the bit string. |

Related Instructions

BT, BTR, BTS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|-----|
| | | | | | | | | U | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BTR Bit Test and Reset

Copies a bit, specified by a zero-based offset index in a register or an 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then clears the bit in the bit string to 0. If the bit base is in a register, the value in the register is the bit string. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the register size) of the bit offset index to select a bit position in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. The instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ for a register offset and 0 to 63 for an immediate value offset.

This instruction is useful for implementing semaphores in concurrent operating systems. Such applications should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-----------------------------|--------------------|--|
| BTR <i>reg/mem16, reg16</i> | 0F B3 | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |
| BTR <i>reg/mem32, reg32</i> | 0F B3 | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |
| BTR <i>reg/mem64, reg64</i> | 0F B3 | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |
| BTR <i>reg/mem16, imm8</i> | 0F BA /6 <i>ib</i> | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |
| BTR <i>reg/mem32, imm8</i> | 0F BA /6 <i>ib</i> | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |
| BTR <i>reg/mem64, imm8</i> | 0F BA /6 <i>ib</i> | Set the carry flag to the value of the selected bit and clear the bit in the bit string to zero. |

Related Instructions

BT, BTC, BTR, BTS

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|-----|
| | | | | | | | | U | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

BTS Bit Test and Set

Copies a bit, specified by a zero-based offset index in a register or an 8-bit immediate value (second operand), from a bit string (first operand), also called the bit base, to the carry flag (CF) of the rFLAGS register, and then sets the bit in the bit string to 1. If the bit base is in a register, the value in the register is the bit string. If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string.

If the bit base operand is a register, the instruction uses the modulo 16, 32, or 64 (depending on the register size) of the bit offset index to select a bit position in the register.

If the bit base operand is a memory location, bit 0 of the byte at the specified address is the bit base of the bit string. The instruction selects a bit position relative to the bit base in the range -2^{63} to $+2^{63} - 1$ for a register offset and 0 to 63 for an immediate value offset.

This instruction is useful for implementing semaphores in concurrent operating systems. Such applications should precede this instruction with the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-----------------------------|--------------------|---|
| BTS <i>reg/mem16, reg16</i> | 0F AB | Set the carry flag to the value of the selected bit and set the bit in the bit string to 1. |
| BTS <i>reg/mem32, reg32</i> | 0F AB | Set the carry flag to the value of the selected bit and set the bit in the bit string to 1. |
| BTS <i>reg/mem64, reg64</i> | 0F AB | Set the carry flag to the value of the selected bit and set the bit in the bit string to 1. |
| BTS <i>reg/mem16, imm8</i> | 0F BA /5 <i>ib</i> | Set the carry flag to the value of the selected bit and set the bit in the bit string to 1. |
| BTS <i>reg/mem32, imm8</i> | 0F BA /5 <i>ib</i> | Sets the carry flag to the value of the selected bit and sets the bit in the bit string to 1. |
| BTS <i>reg/mem64, imm8</i> | 0F BA /5 <i>ib</i> | Set the carry flag to the value of the selected bit and set the bit in the bit string to 1. |

Related Instructions

BT, BTC, BTR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|-----|
| | | | | | | | | U | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CALL Procedure Call

Pushes procedure linking information onto the stack and branches to the target address, which contains the first instruction of the called procedure. The target operand can specify a register, a memory location, or an immediate value. The instruction can execute one of four types of calls: near, far, inter-privilege-level far, or task switch.

| Mnemonic | Opcode | Description |
|-----------------------|--------------|---|
| CALL <i>rel16off</i> | E8 <i>iw</i> | Near call to code in the current code segment at 16-bit address specified by <i>rel16off</i> . (Cannot be encoded in 64-bit mode.) |
| CALL <i>rel32off</i> | E8 <i>id</i> | Near call to code in the current code segment at 32-bit address specified by <i>rel32off</i> . |
| CALL <i>reg/mem16</i> | FF /2 | Near call to address specified in the memory location specified by <i>reg/mem16</i> . (Cannot be encoded in 64-bit mode.) |
| CALL <i>reg/mem32</i> | FF /2 | Near call to address specified in the memory location specified by <i>reg/mem32</i> . |
| CALL <i>reg/mem64</i> | FF /2 | Near call to address specified in the memory location specified by <i>reg/mem64</i> . |
| CALL <i>pntr16:16</i> | 9A <i>cd</i> | Absolute far call. (Invalid in 64-bit mode.) |
| CALL <i>pntr16:32</i> | 9A <i>cp</i> | Absolute far call. (Invalid in 64-bit mode.) |
| CALL <i>mem16:16</i> | FF /3 | Absolute indirect far call. |
| CALL <i>mem16:32</i> | FF /3 | Absolute indirect far call. |

A procedure accessed by a near CALL is located in the same code segment as the CALL instruction. The content of the rIP register (which is the address of the instruction after the CALL) is pushed onto the stack and rSP is decremented by the appropriate number of bytes. Then rIP is loaded with the new value, which is the offset of the near procedure.

At the end of the procedure, RET is used to return control to the instruction after the call. When RET is executed, the rIP is popped off the stack, which returns control to the instruction after the CALL. Direct, register indirect, or memory indirect addressing can be used to code the address of the near procedure.

For near calls in 64-bit mode, the operand size defaults to 64 bits, the E8 opcode results in $RIP = RIP + 32\text{-bit displacement sign-extended to 64 bits}$, and the FF /2 opcode results in $RIP = 64\text{-bit offset from register or memory}$. (The 16-bit displacement size cannot be encoded, because operand size is fixed at 64 bits.)

When calling a far procedure (the procedure is in a different segment from the CALL instruction), the rSP is decremented after the CS:rIP of the instruction following the CALL is pushed onto the stack. CS:rIP is then loaded with the segment and offset address of the called procedure. When pushing CS:rIP onto the stack, CS is pushed first and then rIP. When the RETF is executed, CS and rIP are restored from the stack and execution continues with the instruction following the CALL.

For far calls in 64-bit mode, the 9A opcode is invalid, and the FF /3 opcode does the following:

- For 32-bit operand size, if the selector points to a gate, then $RIP = \text{zero-extended 32-bit offset from the gate}$, else $RIP = \text{zero-extended 32-bit offset from the far pointer referenced in instruction}$.
- For 64-bit operand size, if the selector points to a gate, then $RIP = 64\text{-bit offset from the gate}$, else $RIP = \text{zero-extended 32-bit offset from the far pointer referenced in instruction}$.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

Action

```
// Far calls (CALLF)
// See “Pseudocode Definitions” on page 46.
```

```
CALLF_START:
```

```
IF (REAL_MODE)
    CALLF_REAL_OR_VIRTUAL
ELSIF (PROTECTED_MODE)
    CALLF_PROTECTED
ELSE // (VIRTUAL_MODE)
    CALLF_REAL_OR_VIRTUAL
```

```
CALLF_REAL_OR_VIRTUAL:
```

```
IF (OPCODE = callf [mem])
{
    temp_RIP = READ_MEM.z [mem]
    temp_CS  = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE = callf direct)
{
    temp_RIP = z-sized offset specified in the instruction,
```

```

        zero-extended to 64 bits
    temp_CS = selector specified in the instruction
}

PUSH.v old_CS
PUSH.v next_RIP

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]

CS.sel = temp_CS
CS.base = temp_CS SHL 4
RIP = temp_RIP
EXIT

```

CALLF_PROTECTED:

```

IF (OPCODE = callf [mem])
{
    temp_offset = READ_MEM.z [mem]
    temp_sel     = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE = callf direct)
    IF (64BIT_MODE)
        EXCEPTION [#UD] // 'callf direct' is illegal in 64-bit mode
    {
        temp_offset = z-sized offset specified in the instruction,
                      zero-extended to 64 bits
        temp_sel     = selector specified in the instruction
    }

temp_desc = READ_DESCRIPTOR (temp_sel, cs_chk)

IF (temp_desc.attr.type = 'available_tss')
    TASK_SWITCH // using temp_sel as the target tss selector
ELSIF (temp_desc.attr.type = 'taskgate')
    TASK_SWITCH // using the tss selector in the task gate as the target tss
ELSIF (temp_desc.attr.type = 'code')
    // if the selector refers to a code descriptor, then
    // the offset we read is the target RIP
{
    temp_RIP = temp_offset
    CS = temp_desc
    PUSH.v old_CS
    PUSH.v next_RIP
    IF ((!64BIT_MODE) && (temp_RIP > CS.limit))
        EXCEPTION [#GP(0)] // temp_RIP can't be non-canonical because
                           // it's a 16- or 32-bit number, zero-extended
                           // to 64 bits
    RIP = temp_RIP
}

```

```

EXIT
}
ELSE // (temp_desc.attr.type = 'callgate')
    // if the selector refers to a call gate, then
    // the target CS and RIP both come from the call gate

    IF (LONG_MODE)
        // the size of the gate controls the size of the stack pushes
        V=8-byte
        // long mode only uses 64-bit call gates, force 8-byte opsize
    ELSIF (temp_desc.attr.type = 'callgate32')
        V=4-byte
        // legacy mode, using a 32-bit call-gate, force 4-byte opsize
    ELSE // (temp_desc.attr.type = 'callgate16')
        V=2-byte
        // legacy mode, using a 16-bit call-gate, force 2-byte opsize

    temp_RIP = temp_desc.offset

    IF (LONG_MODE) // in long mode, we need to read the 2nd half of a
        // 16-byte call-gate from the gdt/ldt, to get the upper
        // 32 bits of the target RIP
        temp_upper = READ_MEM.q [temp_sel+8]
        IF (temp_upper.[44:40] != 0)
            EXCEPTION [#GP(0)]
            // make sure the extended attribute bits are all zero
        temp_RIP = temp_RIP + (temp_upper SHL 32)
        // concatenate both halves of RIP

    CS = READ_DESCRIPTOR (temp_desc.segment, clg_chk)

    temp_CPL = CS.sel.rpl
    IF (CPL=temp_CPL)
    {
        PUSH.v old_CS
        PUSH.v next_RIP
    }

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
    {
        EXCEPTION #GP(0)
    }

    RIP = temp_RIP
    EXIT

ELSE // (CPL != temp_CPL), changing privilege
{
    CPL = temp_CPL
    temp_ist = 0 // call-far doesn't use ist pointers
    temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER (CPL, temp_ist)

```

```

    RSP.q = temp_RSP
    SS = temp_SS_desc
    PUSH.v old_SS          // SS on following pushes use ss.sel as error code
    PUSH.v old_RSP
    IF (LEGACY_MODE)       // legacy-mode call gates have a param_count field
        temp_PARAM_COUNT = temp_desc.attr.param_count

        FOR (I=temp_PARAM_COUNT; I>0; I--)
        {
            temp_DATA = READ_MEM.v [SS:old_RSP+I*V]
            PUSH.v temp_DATA
        }

    PUSH.v old_CS
    PUSH.v next_RIP
    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
    {
        EXCEPTION [#GP(0)]
    }
    RIP = temp_RIP
    EXIT

```

Related Instructions

RET

rFLAGS Affected

None, except if task switch occurs, in which case all flags are affected.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | | | X | The absolute CALL far (9A) opcode of this instruction is not recognized in 64-bit mode. |
| Invalid TSS, #TS (selector) | | | X | The target stack segment and ESP pointed to a location that is past the end of the TSS. |
| | | | X | The selector for the target stack segment was null. |
| | | | X | The new stack-segment selector in the TSS contained an RPL that was not equal to the code-segment descriptor's DPL. |
| | | | X | The target stack-segment descriptor contained a DPL that is not equal to that of the code-segment descriptor. |
| | | | X | The target stack segment was not a writable data segment. |
| Stack, #SS | | | X | The index in the stack-segment selector was beyond the descriptor table limits. |
| | | | X | The accessed code segment, call gate, task gate, or TSS was not present. |
| Segment not present, #NP (selector) | | | X | The accessed code segment, call gate, task gate, or TSS was not present. |
| Stack, #SS | | | X | The stack segment limit was exceeded when a stack switch occurred. |
| | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | | X | The stack segment limit was exceeded when a stack switch occurred. |
| | | | X | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |
| | | | X | When the stack switch occurred, there was not enough space in the stack segment for the return address, the parameters, or the stack-segment pointer. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | X | X | X | The offset for the destination operand was outside the boundaries of the target code segment. |
| | | | X | A null data segment was being used to reference memory. |
| | | | X | The the destination operand's segment selector was null. |
| | | | X | The gate contains a code-segment selector that was null. |
| General protection, #GP (selector) | | | X | The index for the code segment, gate, or TSS selector was beyond the limits of the descriptor table. |
| | | | X | The segment descriptor was not a nonconforming-code segment, conforming-code segment, task gate, call gate, or TSS. |
| | | | X | The segment selector's RPL was greater than the CPL, or the nonconforming-code segment's DPL was not equal to the CPL. |
| | | | X | A conforming-code segment's DPL was greater than the CPL. |
| | | | X | A call-gate, task-gate, or TSS descriptor's DPL was less than the CPL, or it was less than the RPL of the call-gate, task-gate, or TSS segment selector. |
| | | | X | The descriptor for a call gate's segment selector did not specify it as a code segment. |
| | | | X | A call gate's segment selector was outside the limits of the descriptor table. |
| | | | X | The code-segment DPL that was specified by a call gate is greater than the CPL. |
| | | | X | The TI bit in the TSS segment selector indicated a local table. |
| | | | X | The segment descriptor for the TSS indicated that the TSS was not available or busy. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CBW**Convert Byte to Word**

Copies the sign (bit 7) of the value in the AL register to all bits of the AH register. The effect of this instruction is to convert a signed byte in the AL register into a signed word in the AX register. This action helps avoid the overflow problem in signed number arithmetic.

| Mnemonic | Opcode | Description |
|-----------------|---------------|-------------------------|
| CBW | 98 | Sign extend AL into AX. |

Related Instructions

CDQ, CWD, CWDE, CDQE

rFLAGS Affected

None

Exceptions

None

CDQ

Convert Doubleword to Quadword

Copies the sign (bit 31) of the value in the EAX register to all bits of the EDX register. The effect of this instruction is to convert a signed doubleword in the EAX register into a signed quadword in the EAX:EDX registers. This action helps avoid the overflow problem in signed number arithmetic.

Use the CDQ instruction to produce a quadword dividend from a doubleword before doubleword division.

The CWD, CDQ and CQ0 mnemonics share opcode 99. The operand-size attribute determines the size of the destination register.

| Mnemonic | Opcode | Description |
|----------|--------|-------------------------------|
| CDQ | 99 | Sign extend EAX into EDX:EAX. |

Related Instructions

CBW, CWD, MOVSX, MOVZX

rFLAGS Affected

None

Exceptions

None

CDQE Convert Doubleword to Quadword (64-bit mode)

Copies the sign (bit 31) of the value in the EAX register to all bits of the RAX register. The effect of this instruction is to convert a signed doubleword in the EAX register into a signed quadword in the RAX register. This action helps avoid the overflow problem in signed number arithmetic.

This mnemonic is meaningful only in 64-bit mode.

| Mnemonic | Opcode | Description |
|-----------------|---------------|---------------------------|
| CDQE | 98 | Sign extend EAX into RAX. |

Related Instructions

CBW, CWDE, CWD, CDQ, CQO

rFLAGS Affected

None

Exceptions

None

CLC Clear Carry Flag

Clears the carry flag (CF) in the rFLAGS register to zero.

| Mnemonic | Opcode | Description |
|----------|--------|------------------------------------|
| CLC | F8 | Clear the carry flag (CF) to zero. |

Related Instructions

STC, CMC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | | | | | | | | 0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

None

CLD Clear Direction Flag

Clears the direction flag (DF) in the rFLAGS register to zero. If the DF flag is 0, each execution of a string instruction increments the data pointer (index registers ESI or EDI). If the DF flag is 1, the string instruction decrements the pointer. Use the CLD instruction before a string instruction to make the data pointer increment.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| CLD | FC | Clear the direction flag (DF) to zero. |

Action

```
directionFlag = 0;
```

Related Instructions

INS_x, LODS_x, MOVS_x, OUTS_x, SCAS_x, STD, STOS_x

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | 0 | | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

None

CLFLUSH**Cache Line Invalidate**

Invalidates the cache line specified by the *mem8* linear-address. The instruction checks all levels of the cache hierarchy—internal caches and external caches—and invalidates the cache line in every cache in which it is found. If a cache contains a dirty copy of the cache line (that is, the cache line is in the *modified* or *owned* MOESI state), the line is written back to memory before it is invalidated. The instruction sets the cache-line MOESI state to *invalid*.

The instruction also checks the physical address corresponding to the linear-address operand against the processor's write-combining buffers. If the write-combining buffer holds data intended for that physical address, the instruction writes the entire contents of the buffer to memory. This occurs even though the data is not cached in the cache hierarchy. In a multiprocessor system, the instruction checks the write-combining buffers only on the processor that executed the CLFLUSH instruction.

The CLFLUSH instruction is weakly-ordered with respect to other instructions that operate on memory. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around a CLFLUSH instruction. Such reordering can cause freshly-loaded cache lines to be flushed unintentionally. The only way to avoid this situation is to use the MFENCE instruction to force strong-ordering of the CLFLUSH instruction with respect to other memory operations. The LFENCE, SFENCE, and serializing instructions are *not* ordered with respect to CLFLUSH.

The CLFLUSH instruction behaves like a load instruction with respect to setting the page-table accessed and dirty bits. That is, it sets the page-table accessed bit to 1, but does not modify the page-table dirty bit.

The CLFLUSH bit (bit 19) returned by the CPUID instruction standard function 1 indicates whether the processor supports the CLFLUSH instruction. The CLFLUSH instruction executes at any privilege level, but it invalidates the cache line only if the segment-protection and page-protection settings permit access for the specified address. Cache lines have the same segment protection as one-byte reads (or loads); if the execute-only bit is set, execute-only code segments are allowed.

| Mnemonic | Opcode | Description |
|--------------------|----------|--|
| CFLUSH <i>mem8</i> | 0F AE /7 | Invalidate cache line at memory location <i>mem8</i> . |

Related Instructions

INVD, WBINVD

rFLAGS Affected

None

Exceptions

| Exception (vector) | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | X | The CLFLUSH instruction is not supported, as indicated by bit 19 of CPUID function 1. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |

CMC Complement Carry Flag

Complements (toggles) the carry flag (CF) bit of the rFLAGS register.

| Mnemonic | Opcode | Description |
|----------|--------|---------------------------------|
| CMC | F5 | Complement the carry flag (CF). |

Action

```
carryFlag = not(carryFlag);
```

Related Instructions

CLC, STC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|-----|
| | | | | | | | | | | | | | | | | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

None

CMOVcc**Conditional Move**

Conditionally moves a 16-bit, 32-bit, or 64-bit value in memory or a general-purpose register (second operand) into a register (first operand), depending upon the settings of condition flags in the rFLAGS register. If the condition is not satisfied, the instruction has no effect and control is passed to the next instruction.

The mnemonics of CMOVcc instructions denote the condition that must be satisfied. Most assemblers provide instruction mnemonics with A (above) and B (below) tags to supply the semantics for manipulating unsigned integers. Those with G (greater than) and L (less than) tags deal with signed integers. Thus, many opcodes may be represented by synonymous mnemonics. For example, the CMOVA instruction is synonymous with the CMOVNBE instruction, and the CMOVE (conditional move if equal) and CMOVZ (conditional move if zero) instructions are aliases and denote the same instruction with the opcode 0F 44h.

Support for CMOVcc instructions depends on the processor implementation. To determine whether a processor can perform CMOVcc instructions, use the CPUID instruction to determine whether bit 15 of CPUID standard function 1 and extended function 8000_0001h is set to 1.

| Mnemonic | Opcode | Description |
|--|---------------|--|
| CMOVA <i>reg16, reg/mem16</i> CMOVNB <i>reg16, reg/mem16</i> | 0F 47 /r | Move if carry flag (CF) and zero flag (ZF) are both 0. |
| CMOVA <i>reg32, reg/mem32</i> CMOVNB <i>reg32, reg/mem32</i> | 0F 47 /r | Move if carry flag (CF) and zero flag (ZF) are both 0. |
| CMOVA <i>reg64, reg/mem64</i> CMOVNB <i>reg64, reg/mem64</i> | 0F 47 /r | Move if carry flag (CF) and zero flag (ZF) are both 0. |
| CMOVAE <i>reg16, reg/mem16</i> CMOVNB <i>reg16, reg/mem16</i> CMOVNC <i>reg16, reg/mem16</i> | 0F 43 /r | Move if carry flag (CF) is 0. |
| CMOVAE <i>reg32, reg/mem32</i> CMOVNB <i>reg32, reg/mem32</i> CMOVNC <i>reg32, reg/mem32</i> | 0F 43 /r | Move if carry flag (CF) is 0. |
| CMOVAE <i>reg64, reg/mem64</i> CMOVNB <i>reg64, reg/mem64</i> CMOVNC <i>reg64, reg/mem64</i> | 0F 43 /r | Move if carry flag (CF) is 0. |

| Mnemonic | Opcode | Description |
|---|---------------|---|
| CMOVB <i>reg16, reg/mem16</i> CMOVC <i>reg16, reg/mem16</i> CMOVNAE <i>reg16, reg/mem16</i> | 0F 42 /r | Move if carry flag (CF) is 1 (below). |
| CMOVB <i>reg32, reg/mem32</i> CMOVC <i>reg32, reg/mem32</i> CMOVNAE <i>reg32, reg/mem32</i> | 0F 42 /r | Move if carry flag (CF) is 1 (below). |
| CMOVB <i>reg64, reg/mem64</i> CMOVC <i>reg64, reg/mem64</i> CMOVNAE <i>reg64, reg/mem64</i> | 0F 42 /r | Move if carry flag (CF) is 1 (below). |
| CMOVBE <i>reg16, reg/mem16</i> CMOVNA <i>reg16, reg/mem16</i> | 0F 46 /r | Move if carry flag (CF) or zero flag (ZF) is 1 (below or equal). |
| CMOVBE <i>reg32, reg/mem32</i> CMOVNA <i>reg32, reg/mem32</i> | 0F 46 /r | Move if carry flag (CF) or zero flag (ZF) is 1 (below or equal). |
| CMOVBE <i>reg64, reg/mem64</i> CMOVNA <i>reg64, reg/mem64</i> | 0F 46 /r | Move if carry flag (CF) or zero flag (ZF) is 1 (below or equal). |
| CMOVE <i>reg16, reg/mem16</i> CMOVZ <i>reg16, reg/mem16</i> | 0F 44 /r | Move if zero flag (ZF) is 1 (equal). |
| CMOVE <i>reg32, reg/mem32</i> CMOVZ <i>reg32, reg/mem32</i> | 0F 44 /r | Move if zero flag (ZF) is 1 (equal/zero). |
| CMOVE <i>reg64, reg/mem64</i> CMOVZ <i>reg64, reg/mem64</i> | 0F 44 /r | Move if zero flag (ZF) is 1 (equal/zero). |
| CMOVG <i>reg16, reg/mem16</i> CMOVNLE <i>reg16, reg/mem16</i> | 0F 4F /r | Move if zero flag (ZF) is zero and the sign flag (SF) is equal to the overflow flag (OF) (greater/not less or equal). |
| CMOVG <i>reg32, reg/mem32</i> CMOVNLE <i>reg32, reg/mem32</i> | 0F 4F /r | Move if zero flag (ZF) is zero and the sign flag (SF) is equal to the overflow flag (OF) (greater/not less or equal). |
| CMOVG <i>reg64, reg/mem64</i> CMOVNLE <i>reg64, reg/mem64</i> | 0F 4F /r | Move if zero flag (ZF) is zero and the sign flag (SF) is equal to the overflow flag (OF) (greater/not less or equal). |
| CMOVGE <i>reg16, reg/mem16</i> CMOVNL <i>reg16, reg/mem16</i> | 0F 4D /r | Move if the sign flag (SF) is equal to the overflow flag (OF). |
| CMOVGE <i>reg32, reg/mem32</i> CMOVNL <i>reg32, reg/mem32</i> | 0F 4D /r | Move if the sign flag (SF) is equal to the overflow flag (OF). |
| CMOVGE <i>reg64, reg/mem64</i> CMOVNL <i>reg64, reg/mem64</i> | 0F 4D /r | Move if the sign flag (SF) is equal to the overflow flag (OF). |
| CMOVL <i>reg16, reg/mem16</i> CMOVNGE <i>reg16, reg/mem16</i> | 0F 4C /r | Move if the sign flag (SF) is not equal to the overflow flag (OF). |

| Mnemonic | Opcode | Description |
|--|---------------|---|
| CMOVL <i>reg32, reg/mem32</i> CMOVNGE <i>reg32, reg/mem32</i> | 0F 4C /r | Move if the sign flag (SF) is not equal to the overflow flag (OF). |
| CMOVL <i>reg64, reg/mem64</i> CMOVNGE <i>reg64, reg/mem64</i> | 0F 4C /r | Move if the sign flag (SF) is not equal to the overflow flag (OF). |
| CMOVLE <i>reg16, reg/mem16</i> CMOVNG <i>reg16, reg/mem16</i> | 0F 4E /r | Move if the zero flag (ZF) is 1 or the sign flag (SF) is not equal to the overflow flag (OF). |
| CMOVLE <i>reg32, reg/mem32</i> CMOVNG <i>reg32, reg/mem32</i> | 0F 4E /r | Move if the zero flag (ZF) is 1 or the sign flag (SF) is not equal to the overflow flag (OF). |
| CMOVLE <i>reg64, reg/mem64</i> CMOVNG <i>reg64, reg/mem64</i> | 0F 4E /r | Move if the zero flag (ZF) is 1 or the sign flag (SF) is not equal to the overflow flag (OF). |
| CMOVNE <i>reg16, reg/mem16</i> | 0F 45 /r | Move if zero flag (ZF) is 0. |
| CMOVNE <i>reg32, reg/mem32</i> | 0F 45 /r | Move if zero flag (ZF) is 0. |
| CMOVNE <i>reg64, reg/mem64</i> | 0F 45 /r | Move if zero flag (ZF) is 0. |
| CMOVNO <i>reg16, reg/mem16</i> | 0F 41 /r | Move if the overflow flag (OF) is 0. |
| CMOVNO <i>reg32, reg/mem32</i> | 0F 41 /r | Move if the overflow flag (OF) is 0. |
| CMOVNO <i>reg64, reg/mem64</i> | 0F 41 /r | Move if the overflow flag (OF) is 0. |
| CMOVNP <i>reg16, reg/mem16</i> | 0F 4B /r | Move if the parity flag (PF) is 0. |
| CMOVNP <i>reg32, reg/mem32</i> | 0F 4B /r | Move if the parity flag (PF) is 0. |
| CMOVNP <i>reg64, reg/mem64</i> | 0F 4B /r | Move if the parity flag (PF) is 0. |
| CMOVNS <i>reg16, reg/mem16</i> | 0F 49 /r | Move if the sign flag (SF) is 0. |
| CMOVNS <i>reg32, reg/mem32</i> | 0F 49 /r | Move if the sign flag (SF) is 0. |
| CMOVNS <i>reg64, reg/mem64</i> | 0F 49 /r | Move if the sign flag (SF) is 0. |
| CMOVNZ <i>reg16, reg/mem16</i> | 0F 45 /r | Move if the zero flag (ZF) is not 0. |
| CMOVNZ <i>reg32, reg/mem32</i> | 0F 45 /r | Move if the zero flag (ZF) is not 0. |
| CMOVNZ <i>reg64, reg/mem64</i> | 0F 45 /r | Move if the zero flag (ZF) is not 0. |
| CMOVO <i>reg16, reg/mem16</i> | 0F 40 /r | Move if the overflow flag (OF) is 1. |
| CMOVO <i>reg32, reg/mem32</i> | 0F 40 /r | Move if the overflow flag (OF) is 1. |
| CMOVO <i>reg64, reg/mem64</i> | 0F 40 /r | Move if the overflow flag (OF) is 1. |
| CMOVP <i>reg16, reg/mem16</i> | 0F 4A /r | Move if the parity flag (PF) is 1. |
| CMOVP <i>reg32, reg/mem32</i> | 0F 4A /r | Move if the parity flag (PF) is 1. |

| Mnemonic | Opcode | Description |
|--------------------------------|-----------------|------------------------------------|
| CMOVP <i>reg64, reg/mem64</i> | 0F 4A/ <i>r</i> | Move if the parity flag (PF) is 1. |
| CMOVPE <i>reg16, reg/mem16</i> | 0F 4A/ <i>r</i> | Move if the parity flag (PF) is 1. |
| CMOVPE <i>reg32, reg/mem32</i> | 0F 4A/ <i>r</i> | Move if the parity flag (PF) is 1. |
| CMOVPE <i>reg64, reg/mem64</i> | 0F 4A/ <i>r</i> | Move if the parity flag (PF) is 1. |
| CMOVPO <i>reg16, reg/mem16</i> | 0F 4B/ <i>r</i> | Move if the parity flag (PF) is 0. |
| CMOVPO <i>reg32, reg/mem32</i> | 0F 4B/ <i>r</i> | Move if the parity flag (PF) is 0. |
| CMOVPO <i>reg64, reg/mem64</i> | 0F 4B/ <i>r</i> | Move if the parity flag (PF) is 0. |
| CMOVS <i>reg16, reg/mem16</i> | 0F 48/ <i>r</i> | Move if the sign flag (SF) is 1. |
| CMOVS <i>reg32, reg/mem32</i> | 0F 48/ <i>r</i> | Move if the sign flag (SF) is 1. |
| CMOVS <i>reg64, reg/mem64</i> | 0F 48/ <i>r</i> | Move if the sign flag (SF) is 1. |

Related Instructions

MOV, MOVD, MOVQ, MOVS, MOVSB, MOVSW, MOVSD, MOVSX, MOVZX

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Invalid opcode, #UD | X | X | X | The CMOVx instruction is not supported, as indicated by bit 15 of CPUID function 1 or extended function 8000_0001. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CMP**Compare**

Compares the contents of a register or memory location (first operand) with an immediate value or the contents of a register or memory location (second operand), and sets or clears the status flags in the rFLAGS register to reflect the results. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. If the second operand is an immediate value, the instruction sign-extends the value to the length of the first operand.

Use the CMP instruction to set the condition codes for a subsequent conditional jump (Jcc), conditional move (CMOVcc), or conditional SETcc instruction. Appendix E, “Instruction Effects on RFLAGS,” shows how instructions affect the rFLAGS status flags.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|--|
| CMP AL, <i>imm8</i> | 3C <i>ib</i> | Compare an 8-bit immediate value with the contents of the AL register. |
| CMP AX, <i>imm16</i> | 3D <i>iw</i> | Compare a 16-bit immediate value with the contents of the AX register. |
| CMP EAX, <i>imm32</i> | 3D <i>id</i> | Compare a 32-bit immediate value with the contents of the EAX register. |
| CMP RAX, <i>imm32</i> | 3D <i>id</i> | Compare a 32-bit immediate value with the contents of the RAX register. |
| CMP <i>reg/mem8</i> , <i>imm8</i> | 80 /7 <i>ib</i> | Compare an 8-bit immediate value with the contents of an 8-bit register or memory operand. |
| CMP <i>reg/mem16</i> , <i>imm16</i> | 81 /7 <i>iw</i> | Compare a 16-bit immediate value with the contents of a 16-bit register or memory operand. |
| CMP <i>reg/mem32</i> , <i>imm32</i> | 81 /7 <i>id</i> | Compare a 32-bit immediate value with the contents of a 32-bit register or memory operand. |
| CMP <i>reg/mem64</i> , <i>imm32</i> | 81 /7 <i>id</i> | Compare a 32-bit immediate value with the contents of a 64-bit register or memory operand. |
| CMP <i>reg/mem16</i> , <i>imm8</i> | 83 /7 <i>ib</i> | Compare the contents of a 16-bit register or memory operand with an 8-bit immediate value. |
| CMP <i>reg/mem32</i> , <i>imm8</i> | 83 /7 <i>ib</i> | Compare the contents of a 32-bit register or memory operand with an 8-bit immediate value. |

| Mnemonic | Opcode | Description |
|-----------------------------|-----------------|---|
| <i>CMP reg/mem64, imm8</i> | 83 /7 <i>ib</i> | Compare the contents of a 64-bit register or memory operand with an 8-bit immediate value. |
| <i>CMP reg/mem8, reg8</i> | 38 / <i>r</i> | Compare the contents of an 8-bit register or memory operand with the contents of an 8-bit register. |
| <i>CMP reg/mem16, reg16</i> | 39 / <i>r</i> | Compare the contents of a 16-bit register or memory operand with the contents of a 16-bit register. |
| <i>CMP reg/mem32, reg32</i> | 39 / <i>r</i> | Compare the contents of a 32-bit register or memory operand with the contents of a 32-bit register. |
| <i>CMP reg/mem64, reg64</i> | 39 / <i>r</i> | Compare the contents of a 64-bit register or memory operand with the contents of a 64-bit register. |
| <i>CMP reg8, reg/mem8</i> | 3A / <i>r</i> | Compare the contents of an 8-bit register with the contents of an 8-bit register or memory operand. |
| <i>CMP reg16, reg/mem16</i> | 3B / <i>r</i> | Compare the contents of a 16-bit register with the contents of a 16-bit register or memory operand. |
| <i>CMP reg32, reg/mem32</i> | 3B / <i>r</i> | Compare the contents of a 32-bit register with the contents of a 32-bit register or memory operand. |
| <i>CMP reg64, reg/mem64</i> | 3B / <i>r</i> | Compare the contents of a 64-bit register with the contents of a 64-bit register or memory operand. |

When comparing unsigned operands, flag settings are as follows:

| Operands | CF | ZF |
|-----------------|-----------|-----------|
| dest > source | 0 | 0 |
| dest = source | 0 | 1 |
| dest < source | 1 | 0 |

When comparing signed operands, flag settings are as follows:

| Operands | ZF | OF |
|-----------------|-----------|-----------|
| dest > source | 0 | SF |
| dest = source | 1 | Undefined |
| dest < source | Undefined | NOT SF |

Related Instructions

SUB, CMPS, CMPSB, CMPSD, CMPSW

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CMPS **Compare Strings**

CMPSB

CMPSW

CMPSD

CMPSQ

Compares the bytes, words, doublewords, or quadwords pointed to by the rSI and rDI registers, sets or clears the status flags of the rFLAGS register to reflect the results, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register. To perform the comparison, the instruction subtracts the second operand from the first operand and sets the status flags in the same manner as the SUB instruction, but does not alter the first operand. The two operands must be the same size.

If the DF flag is 0, the instruction increments rSI and rDI; otherwise, it decrements the pointers. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the CMPSx instruction with explicit operands address the first operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix. These instructions always address the second operand at ES:[rDI] (ES may not be overridden). The explicit operands serve only to specify the type (size) of the values being compared.

The no-operands forms of the instruction use the DS:[rSI] and ES:[rDI] registers to point to the values to be compared (they do not allow a segment prefix). The mnemonic determines the size of the operands and the specific pointer registers used.

Do not confuse this CMPSD instruction with the same-mnemonic CMPSD (compare scalar double-precision floating-point) instruction in the 128-bit media instruction set. Assemblers can distinguish the instructions by the number and type of operands.

For block comparisons, the CMPS instruction supports the REPE or REPZ prefixes (they are synonyms) and the REPNE or REPNZ prefixes (they are synonyms). For details about the REP prefixes, see “Repeat Prefixes” on page 10. If a conditional jump instruction like JL follows a CMPSx instruction, the jump occurs if the value of the *seg*:[rSI] operand is less than the ES:[rDI] operand. This action allows lexicographical comparisons of string or array elements. A CMPSx instruction can also operate inside a loop controlled by the LOOPcc instruction.

| Mnemonic | Opcode | Description |
|--------------------------|--------|---|
| CMPS <i>mem8, mem8</i> | A6 | Compare the byte at DS:SI with the byte at ES:DI and set flags in RFLAGS to reflect the result. |
| CMPS <i>mem16, mem16</i> | A7 | Compare the word at DS:SI with the word at ES:DI and set flags in RFLAGS to reflect the result. |
| CMPS <i>mem32, mem32</i> | A7 | Compare the doubleword at DS:ESI with the doubleword at ES:EDI and set flags in RFLAGS to reflect the result. |
| CMPS <i>mem64, mem64</i> | A7 | Compare the quadword at DS:RSI with the quadword at ES:RDI and set flags in RFLAGS to reflect the result. |
| CMPSB | A6 | Compare the byte at DS:SI with the byte at ES:DI and set flags in RFLAGS to reflect the result. |
| CMPSW | A7 | Compare the word at DS:SI with the word at ES:DI and set flags in RFLAGS to reflect the result. |
| CMPSD | A7 | Compare the doubleword at DS:ESI with the doubleword at ES:EDI and set flags in RFLAGS to reflect the result. |
| CMPSQ | A7 | Compare the quadword at DS:RSI with the quadword at ES:RDI and set flags in RFLAGS to reflect the result. |

Related Instructions

CMP, CMPSB, CMPSW, CMPSD

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CMPXCHG Compare and Exchange

Compares the value in the AL, AX, EAX, or RAX register (second operand) with the value in another register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the second operand and clears the ZF flag to 0.

The memory-operand variants of the CMPXCHG instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|---------------------------------|----------|---|
| CMPXCHG <i>reg/mem8, reg8</i> | 0F B0 /r | Compare AL register to 8-bit register or memory location. If equal, set the zero flag (ZF) to 1 and copy the AL register to the other register or memory location. Otherwise, copy the other register or memory location into AL. |
| CMPXCHG <i>reg/mem16, reg16</i> | 0F B1 /r | Compare AX register to 16-bit register or memory location. If equal, set the zero flag (ZF) to 1 and copy the AX register to the other register or memory location. Otherwise, copy the other register or memory location to AX. |
| CMPXCHG <i>reg/mem32, reg32</i> | 0F B1 /r | Compare EAX register to 32-bit register or memory location. If equal, set the zero flag (ZF) to 1 and copy the EAX register to the other register or memory location. Otherwise, copy the other register or memory location to EAX. |
| CMPXCHG <i>reg/mem64, reg64</i> | 0F B1 /r | Compare RAX register to 64-bit register or memory location. If equal, set the zero flag (ZF) to 1 and copy the RAX register to the other register or memory location. Otherwise, copy the other register or memory location to RAX. |

Related Instructions

CMPXCHG8B

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CMPXCHG8B**Compare and Exchange Eight Bytes**

Compares a 64-bit value in the EDX:EAX register with a 64-bit value in the specified memory location. If the values are equal, the instruction copies the value in the ECX:EBX register to the memory location and sets the zero flag (ZF) of the rFLAGS register to 1. Otherwise, it copies the value in memory to the EDX:EAX register and clears ZF to 0.

The CMPXCHG8B instruction supports the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

Support for the CMPXCHG8B instruction depends on the processor implementation. To find out if a processor can execute the CMPXCHG8B instruction, use the CPUID instruction to determine whether bit 8 of CPUID standard function 1 or extended function 8000_0001h is set to 1.

| Mnemonic | Opcode | Description |
|------------------------|---------------------|---|
| CMPXCHG8B <i>mem64</i> | 0F C7 /1 <i>m64</i> | Compare EDX:EAX register to 64-bit memory location. If equal, set the zero flag (ZF) to 1 and copy the EDX:EAX register to the memory location. Otherwise, copy the memory location to RAX. |

Related Instructions

CMPXCHG

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | X | The CMPXCH8B instruction is not supported, as indicated by bit 8 of CPUID function 1 or extended function 8000_0001. Destination operand was a register. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CPUID

Processor Identification

Provides information about the processor and its capabilities through a number of different functions. Software should load the number of the CPUID function to execute into the EAX register before executing the CPUID instruction. The processor returns information in the EAX, EBX, ECX, and EDX registers; the contents and format of these registers depend on the function.

The architecture supports CPUID information about *standard functions* and *extended functions*. The standard functions have numbers in the 0000_XXXXh series (for example, function 1). To determine the largest standard function number that a processor supports, execute CPUID function 0.

The extended functions have numbers in the 8000_XXXXh series (for example, function 8000_0001h). To determine the largest extended function number that a processor supports, execute CPUID function 8000_0000h. If the value returned in EAX is greater than 8000_0000h, the processor supports extended functions.

Software operating at any privilege level can execute the CPUID instruction to collect this information. In 64-bit mode, this instruction works the same as in legacy mode except that, if the operand size is 32 or 64 bits, it zero-extends 32-bit register results to 64 bits.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| CPUID | 0F A2 | Executes the CPUID function whose number is in the EAX register. |

Testing for the CPUID Instruction

To avoid an invalid-opcode exception (#UD) on those processor implementations that do not support the CPUID instruction, software must first test to determine if the CPUID instruction is supported. Support for the CPUID instruction is indicated by the ability to write the ID bit in the rFLAGS register. Normally, 32-bit software uses the PUSHFD and POPFD instructions in an attempt to write rFLAGS.ID. After reading the updated rFLAGS.ID bit, a comparison determines if the operation changed its value. If the value changed, the processor executing the code supports the CPUID instruction. If the value did not change, rFLAGS.ID is not writable, and the processor does not support the CPUID instruction.

The following code sample shows how to test for the presence of the CPUID instruction using 32-bit code.

```

pushfd                ; save EFLAGS
pop                   eax    ; store EFLAGS in EAX
mov                   ebx, eax ; save in EBX for later testing
xor                   eax, 00200000h ; toggle bit 21
push                  eax    ; push to stack
popfd                 ; save changed EAX to EFLAGS
pushfd                ; push EFLAGS to TOS
pop                   eax    ; store EFLAGS in EAX
cmp                   eax, ebx ; see if bit 21 has changed
jz                    NO_CPUID ; if no change, no CPUID

```

Function 0: Processor Vendor and Largest Standard Function Number

All software using the CPUID instruction must execute function 0. This function returns the largest standard function number and the processor vendor.

EAX: Largest Standard Function Number. Function 0 loads EAX with the largest CPUID standard function number supported by the processor implementation.

EBX, EDX, and ECX: Processor Vendor. Function 0 loads a 12-character string into the EBX, EDX, and ECX registers identifying the processor vendor. For AMD processors, the string is `AuthenticAMD`. This string informs software that it should follow the AMD CPUID definition for subsequent CPUID function calls. If the function returns a another vendor's string, software must use that vendor's CPUID definition when interpreting the results of subsequent CPUID function calls. Table 3-1 shows the contents of the EBX, EDX, and ECX registers after executing function 0 on an AMD processor.

Table 3-1. Processor Vendor Return Values

| Register | Return Value | ASCII Characters |
|----------|--------------|------------------|
| EBX | 6874_7541h | "h t u A" |
| EDX | 6974_6E65h | "i t n e" |
| ECX | 444D_4163h | "D M A c" |

Function 1: Processor Signature and Standard Features

Function 1 returns the processor signature and standard-feature bits.

EAX: Processor Signature. Function 1 returns the processor signature in the EAX register; the signature provides information on the processor revision (stepping) level and processor model, as well as the instruction family that the processor supports.

Figure 3-1 shows the format of the EAX register following execution of CPUID function 1.

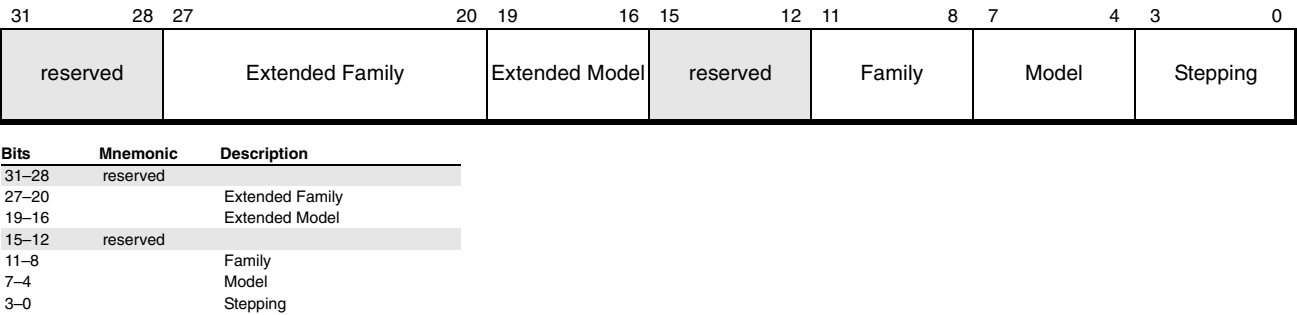


Figure 3-1. Processor Signature (EAX Register)

The Extended Family and Extended Model fields extend the Family and Model fields, respectively, to accommodate larger family and model values. The method for computing the actual—or *effective*—family and model depends on the value of the Family field. The method for computing the effective family is shown in Table 3-2.

Table 3-2. Effective Family Computation

| Family Field | How to Compute the Effective Family | Example |
|--------------|---|---------|
| Fh | Add the Extended Family field and the zero-extended Family field. | |
| Less than Fh | Use the Family field as the effective family. | |

The method for computing the effective model is shown in Table 3-3 on page 116.

Table 3-3. Effective Model Computation

| Family Field | How to Compute the Effective Model | Example |
|--------------|---|---------|
| Fh | Shift the Extended Model field four bits to the left and add it to the Model field. | |
| Less than Fh | Use the Model field as the effective model. | |

EBX: Initial APIC ID, CLFLUSH Size, and Brand ID. Function 1 returns information on the initial value of the physical ID register associated with the advanced programmable interrupt controller (APIC), the size of the cache line flushed by the CLFLUSH instruction, and the processor brand.

Figure 3-2 shows the format of the EBX register following execution of CPUID function 1.

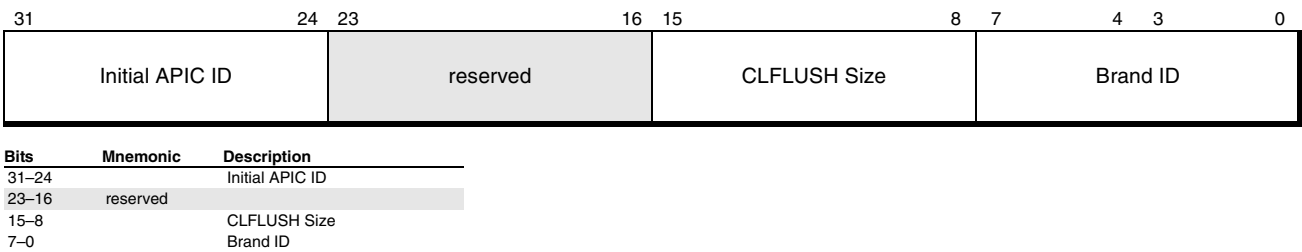


Figure 3-2. Initial APIC ID, CLFLUSH Size, and Brand ID (EBX Register)

The Initial APIC ID field contains the initial value of the processor’s local APIC physical ID register. This value is composed of the Northbridge NodeID (bits 26–24) and the CPU number within the node (bits 31–27). Subsequent writes by software to

the local APIC physical ID register do not change the value of the Initial APIC ID field.

The CLFLUSH Size field gives the size (in quadwords) of the cache line that is flushed by the CLFLUSH instruction. This field is implemented only if the CLFLUSH instruction is supported. To determine if the CLFLUSH instruction is supported, test the CLFLUSH Instruction bit provided by function 8000_0001h.

The Brand ID field identifies a processor with a unique set of features as a specific brand. The BIOS uses the Brand ID field to program the processor name string that is returned by functions 8000_0002h–8000_0004h. If the Brand ID field is 0, the processor does not support the Brand ID feature.

ECX. For function 1, the ECX register is reserved.

EDX: Standard Feature Support. Function 1 returns standard-feature bits in the EDX register. The value of each bit indicates whether support for a specific feature is present on the processor implementation. If the value of a bit is 1, the feature is supported. If the value is 0, the feature is not supported.

Table 3-4 summarizes the standard-feature bits returned in the EDX register for function 1.

Table 3-4. CPUID Standard Feature Support (Function 1)

| EDX Bit | Feature (feature is supported if bit is set to 1) |
|---------|--|
| 0 | On-Chip x87-Instruction Unit. |
| 1 | Virtual-Mode Extensions. See “Virtual Interrupts” in Volume 2. |
| 2 | Debugging Extensions. See “Software-Debug Resources” in Volume 2. |
| 3 | Page-Size Extensions (PSE). See “Page-Size Extensions (PSE) Bit” in Volume 2. |
| 4 | Time-Stamp Counter. See “Time-Stamp Counter” in Volume 2. |
| 5 | AMD K86 Model-Specific Registers (MSRs), with RDMSR and WRMSR Instructions. See “Model-Specific Registers (MSRs)” in Volume 2. |
| 6 | Physical-Address Extensions (PAE). See “Physical-Address Extensions (PAE) Bit” in Volume 2. |
| 7 | Machine Check Exception. See “Handling Machine Check Exceptions” in Volume 2. |
| 8 | CMPXCHG8B Instruction. |
| 9 | Advanced Programmable Interrupt Controller (APIC). BIOS must enable the local APIC. See the documentation for particular implementations of the architecture. |

Table 3-4. CPUID Standard Feature Support (Function 1) (continued)

| EDX Bit | Feature (feature is supported if bit is set to 1) |
|---------|---|
| 10 | Reserved. |
| 11 | SYSENTER and SYSEXIT Instructions. These instructions have different implementations than the SYSCALL and SYSRET instructions indicated by bit 11 of function 8000_0001h. See “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2. |
| 12 | Memory-Type Range Registers (MTRRs). See “Memory-Type Range Registers” in Volume 2. |
| 13 | Page Global Extension. See “Global-Pages” in Volume 2. |
| 14 | Machine Check Architecture. See “Machine Check Mechanism” in Volume 2. |
| 15 | Conditional Move Instructions. Indicates support for conditional move (CMOVcc) general-purpose instructions, and—if the on-chip x87-instruction-unit bit (bit 0) is also set—for the x87 floating-point conditional move (FCMOVcc) instructions. |
| 16 | Page Attribute Table (PAT). See “Memory-Type Range Registers” in Volume 2. |
| 17 | Page-Size Extensions (PSE). See “Page-Size Extensions (PSE) Bit” in Volume 2. |
| 18 | Reserved. |
| 19 | CLFLUSH Instruction. Indicates support for the CLFLUSH (writeback, if modified, and invalidate) general-purpose instruction. |
| 20–22 | Reserved. |
| 23 | MMX™ Instructions. Indicates support for the integer (MMX) 64-bit media instructions. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” |
| 24 | FXSAVE and FXRSTOR Instructions. See “FXSAVE and FXRSTOR Instructions” in Volume 2. |
| 25 | SSE Instructions. Indicates support for the SSE instructions, except that the SSE instructions indicated for the AMD Extensions to MMX Instructions feature (bit 22 of function 8000_0001h; see Table 3-5 on page 120) are implemented if bit 25 is cleared and bit 22 of function 8000_0001h is set. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” |
| 26 | SSE2 Instruction Extensions. Indicates support for the SSE2 instructions. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” |
| 27–31 | Reserved. |

Function 8000_0000h: Processor Vendor and Largest Extended Function Number

Function 8000_0000h mimics the behavior of function 0, except that function 8000_0000h returns the largest *extended* function number instead of the largest standard function number.

EAX: Largest Extended Function Number. Function 8000_0000h loads EAX with the largest CUID extended function number supported by the processor implementation.

EBX, EDX, and ECX: Processor Vendor. Function 8000_0000h loads a 12-character string into the EBX, EDX, and ECX registers identifying the processor vendor. For AMD processors, the string is AuthenticAMD. This string informs software that it should follow the AMD CUID definition for subsequent CUID function calls. If the function returns a another vendor's string, software must use that vendor's CUID definition when interpreting the results of subsequent CUID function calls. Table 3-1 on page 114 shows the contents of the EBX, EDX, and ECX registers after executing function 8000_0000h on an AMD processor.

Function 8000_0001h: Processor Signature and AMD Features

Like function 1, function 8000_0001h returns the processor signature and feature bits. However, the feature bits returned by this function include a subset of the bits reported by function 1, along with additional bits for AMD features.

EAX: Processor Signature. Function 8000_0001h returns the processor signature in the EAX register; the signature provides information on the processor revision (stepping) level and processor model, as well as the instruction family that the processor supports.

Figure 3-1 on page 115 shows the format of the EAX register following execution of CUID function 8000_0001h. (The value returned in the EAX register for function 8000_0001h is the same as the value returned by function 1.)

EBX and ECX. For function 8000_0001h, the EBX and ECX registers are reserved.

EDX: AMD Feature Support. Function 8000_0001h returns information about AMD features—those features that were originally implemented by AMD—in the EDX register. The value of each bit indicates whether support for a specific feature is present on the processor implementation. If the value of a bit is 1, the feature is supported. If the value is 0, the feature is not supported.

Function 8000_0001h also duplicates some of the standard-feature bits from function 1 in the EDX register, but this practice is outdated. Any new feature that is first implemented by a given vendor is now reported only by a function assigned to that vendor.

Table 3-5 summarizes the feature bits returned in the EDX register for function 8000_0001h. The right-most column of this table indicates whether a given bit has the same meaning in function 1. If the bit has the same meaning, use function 1 to test whether the feature is supported. For a list of the feature bits returned by function 1, see Table 3-4 on page 117.

Table 3-5. CPUID AMD Feature Support (Function 8000_0001h)

| EDX Bit | Feature (feature is supported if bit is set to 1) | Same as Function 1 (Table 3-4)¹ |
|--|---|---|
| 0 | On-Chip x87-Instruction Unit. | yes |
| 1 | Virtual-Mode Extensions. See “Virtual Interrupts” in Volume 2. | yes |
| 2 | Debugging Extensions. See “Software-Debug Resources” in Volume 2. | yes |
| 3 | Page-Size Extensions (PSE). See “Page-Size Extensions (PSE) Bit” in Volume 2. | yes |
| 4 | Time-Stamp Counter. See “Time-Stamp Counter” in Volume 2. | yes |
| 5 | AMD K86 Model-Specific Registers (MSRs), with RDMSR and WRMSR Instructions. See “Model-Specific Registers (MSRs)” in Volume 2. | yes |
| 6 | Physical-Address Extensions (PAE). See “Physical-Address Extensions (PAE) Bit” in Volume 2. | yes |
| 7 | Machine Check Exception. See “Handling Machine Check Exceptions” in Volume 2. | yes |
| 8 | CMPXCHG8B Instruction. | yes |
| 9 | Advanced Programmable Interrupt Controller (APIC). BIOS must enable the local APIC. See the documentation for particular implementations of the architecture. | yes |
| 10 | Reserved. | no |
| 11 | SYSCALL and SYSRET Instructions. These instructions have different implementations than the SYSENTER and SYSEXIT instructions indicated by bit 11 of function 1. For additional information, see “Fast System Call and Return” in Volume 2. | no |
| 12 | Memory-Type Range Registers (MTRRs). See “Memory-Type Range Registers” in Volume 2. | yes |
| 13 | Page Global Extension. See “Global-Pages” in Volume 2. | yes |
| 14 | Machine Check Architecture. See “Machine Check Mechanism” in Volume 2. | yes |
| 15 | Conditional Move Instructions. Indicates support for conditional move (CMOVcc) general-purpose instructions, and—if the on-chip x87-instruction-unit bit (bit 0) is also set—for the x87 floating-point conditional move (FCMOVcc) instructions. | yes |
| 16 | Page Attribute Table (PAT). See “Memory-Type Range Registers” in Volume 2. | yes |
| 17 | Page-Size Extensions (PSE). See “Page-Size Extensions (PSE) Bit” in Volume 2. | yes |
| Note: 1. If a bit has the same meaning for function 1 as it does for function 8000_0001h, the processor sets or clears the bit identically for both functions. | | |

Table 3-5. CPUID AMD Feature Support (Function 8000_0001h) (continued)

| EDX Bit | Feature (feature is supported if bit is set to 1) | Same as Function 1 (Table 3-4)¹ |
|--|--|---|
| 18–19 | Reserved. | no |
| 20 | No-Execute Page Protection. See “No Execute (NX) Bit” in Volume 2. | no |
| 21 | Reserved. | no |
| 22 | AMD Extensions to MMX™ Instructions. Indicates support for the AMD extensions to the integer (MMX) 64-bit media instructions, including support for certain SSE and SSE2 instructions. See Appendix D, “Instruction Subsets and CPUID Feature Sets,” for details. | no |
| 23 | MMX™ Instructions. Indicates support for the integer (MMX) 64-bit media instructions. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” | yes |
| 24 | FXSAVE and FXRSTOR Instructions. See “FXSAVE and FXRSTOR Instructions” in Volume 2. | yes |
| 25–28 | Reserved. | no |
| 29 | Long Mode. See “Long Mode” in Volume 2. | no |
| 30 | AMD Extensions to 3DNow!™ Instructions. Indicates support for the AMD extensions to the floating-point (3DNow!) 64-bit media instructions. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” | no |
| 31 | AMD 3DNow!™ Instructions. Indicates support for the floating-point (3DNow!) 64-bit media instructions. For details, see Appendix D, “Instruction Subsets and CPUID Feature Sets.” | no |
| Note: 1. If a bit has the same meaning for function 1 as it does for function 8000_0001h, the processor sets or clears the bit identically for both functions. | | |

Functions 8000_0002h–8000_0004h: Processor Name

Functions 8000_0002h, 8000_0003h, and 8000_0004h together return an ASCII string containing the name of the processor implementation. Software can simply call these three functions in numerical order to obtain a 48-character ASCII name string. Although the name string can be up to 48 characters in length, shorter names have unused byte locations filled with the ASCII null character (00h).

Note: The BIOS must program the name string before these functions are executed; otherwise, these functions return the default processor name string (48 ASCII null characters).

The name string returned by these functions is in little-endian format. Function 8000_0002h returns the first 16 characters of the name and function 8000_0004h returns the last 16 characters. For each of the three groups of 16

characters, the functions return the name (in order of least-significant to most-significant byte) in the EAX, EBX, ECX, and EDX registers. The first character resides in the least-significant byte of EAX, and the last character resides in the most-significant byte of EDX.

Table 3-6 on page 122 gives an example of the return values and their equivalent ASCII characters for a processor with the following name string:

AMD Athlon(tm) processor

Table 3-6. Processor Name String Example

| Function | Register | Return Value | ASCII Characters |
|------------|----------|--------------|------------------|
| 8000_0002h | EAX | 2044_4D41h | "space D M A" |
| | EBX | 6C68_7441h | "l h t A" |
| | ECX | 7428_6E6Fh | "t (n o" |
| | EDX | 7020_296Dh | "p space) m" |
| 8000_0003h | EAX | 6563_6F72h | "e c o r" |
| | EBX | 726F_7373h | "r o s s" |
| | ECX | 0000_0000h | |
| | EDX | 0000_0000h | |
| 8000_0004h | EAX | 0000_0000h | |
| | EBX | 0000_0000h | |
| | ECX | 0000_0000h | |
| | EDX | 0000_0000h | |

Functions 8000_0005h and 8000_0006h: Cache Information

The CPUID functions 8000_0005h and 8000_0006h provide cache and TLB information. These functions are useful to diagnostic software that displays information about the system and the configuration of the processor implementation, including cache size and organization. For more information about the TLB and on-chip caches, see "Translation-Lookaside Buffer (TLB)" in Volume 2 and "Memory Caches" in Volume 2.

Function 8000_0005h returns information about the TLBs and L1 caches integrated on the processor. Tables 3-7, 3-8, 3-9, and 3-10 show the register formats for the information returned by function 8000_0005h.

In these tables, the associativity field is encoded as follows:

- 00h—Reserved.
- 01h—Direct mapped.
- 02h through FEh—The value represents the actual associativity. For example, a value of 04h indicates 4-way associativity.
- FFh—Fully associative.

Table 3-7. CPUID TLB Bits for 2-Mbyte and 4-Mbyte Pages

| Register | Data TLB | | Instruction TLB | |
|---|---------------|--------------------------------|-----------------|--------------------------------|
| | Associativity | Number of Entries ¹ | Associativity | Number of Entries ¹ |
| EAX | Bits 31–24 | Bits 23–16 | Bits 15–8 | Bits 7–0 |
| Note: 1. The number of entries returned is the number of entries available for the 2-Mbyte page size. The 4-Mbyte pages require two 2-Mbyte entries, so the number of entries available for the 4-Mbyte page size is one-half the returned value. | | | | |

Table 3-8. CPUID TLB Bits for 4-Kbyte Pages

| Register | Data TLB | | Instruction TLB | |
|----------|---------------|-------------------|-----------------|-------------------|
| | Associativity | Number of Entries | Associativity | Number of Entries |
| EBX | Bits 31–24 | Bits 23–16 | Bits 15–8 | Bits 7–0 |

Table 3-9. CPUID L1 Data Cache Bits

| Register | L1 Data Cache | | | |
|----------|---------------|---------------|---------------|-------------------|
| | Size (Kbytes) | Associativity | Lines Per Tag | Line Size (Bytes) |
| ECX | Bits 31–24 | Bits 23–16 | Bits 15–8 | Bits 7–0 |

Table 3-10. CPUID L1 Instruction Cache Bits

| Register | L1 Instruction Cache | | | |
|----------|----------------------|---------------|---------------|-------------------|
| | Size (Kbytes) | Associativity | Lines Per Tag | Line Size (Bytes) |
| EDX | Bits 31–24 | Bits 23–16 | Bits 15–8 | Bits 7–0 |

Function 8000_0006h returns information about the L2 cache integrated on the processor. Tables 3-11, 3-12, and 3-13 show the register-content formats for the information returned by function 8000_0006h.

In these tables, the associativity field is encoded as follows:

- 00h—The L2 cache is off (disabled).
- 01h—Direct mapped.
- 02h—2-way associative.
- 04h—4-way associative.
- 06h—8-way associative.
- 08h—16-way associative.
- 0Fh—Fully associative.
- All other encodings are reserved.

Table 3-11. CPUID L2 TLB Bits for 2-Mbyte and 4-Mbyte Pages

| Register | L2 Data TLB | | L2 Instruction or Unified L2 TLB ¹ | |
|--|---------------|--------------------------------|---|--------------------------------|
| | Associativity | Number of Entries ² | Associativity | Number of Entries ² |
| EAX | Bits 31–28 | Bits 27–16 | Bits 15–12 | Bits 11–0 |
| Note: <ol style="list-style-type: none"> 1. The presence of a unified L2 TLB is indicated by a value of 0000h in the upper 16 bits of the EAX register. The unified L2 TLB information is contained in the lower 16 bits of the EAX register. 2. The number of entries returned is the number of entries available for the 2-Mbyte page size. The 4-Mbyte pages require two 2-Mbyte entries, so the number of entries available for the 4-Mbyte page size is one-half the returned value. | | | | |

Table 3-12. CPUID L2 TLB Bits for 4-Kbyte Pages

| Register | L2 Data TLB | | L2 Instruction or Unified L2 TLB ¹ | |
|---|---------------|-------------------|---|-------------------|
| | Associativity | Number of Entries | Associativity | Number of Entries |
| EBX | Bits 31–28 | Bits 27–16 | Bits 15–12 | Bits 11–0 |
| Note: <ol style="list-style-type: none"> 1. The presence of a unified L2 TLB is indicated by a value of 0000h in the upper 16 bits of the EBX register. The unified L2 TLB information is contained in the lower 16 bits of the EBX register. | | | | |

Table 3-13. CPUID L2 Cache Bits

| Register | L2 Cache | | | |
|----------|---------------|---------------|---------------|-------------------|
| | Size (Kbytes) | Associativity | Lines Per Tag | Line Size (Bytes) |
| ECX | Bits 31–16 | Bits 15–12 | Bits 11–8 | Bits 7–0 |

EDX. For function 8000_0006h, the EDX register is reserved.

Function 8000_0007h: Advanced Power Management Features

Function 8000_0007h returns information about the advanced-power-management features supported by the processor.

EAX, EBX, and ECX. For function 8000_0007h, the EAX, EBX, and ECX registers are reserved.

EDX. Function 8000_0007h returns information about advanced-power-management features in the EDX register. Figure 3-3 shows the format of the EDX register following execution of CPUID function 8000_0007h. Each bit indicates whether support for a specific feature is present on the processor implementation. If the value of a power-management-feature bit is 1, the feature is supported. If the value is 0, the feature is not supported.

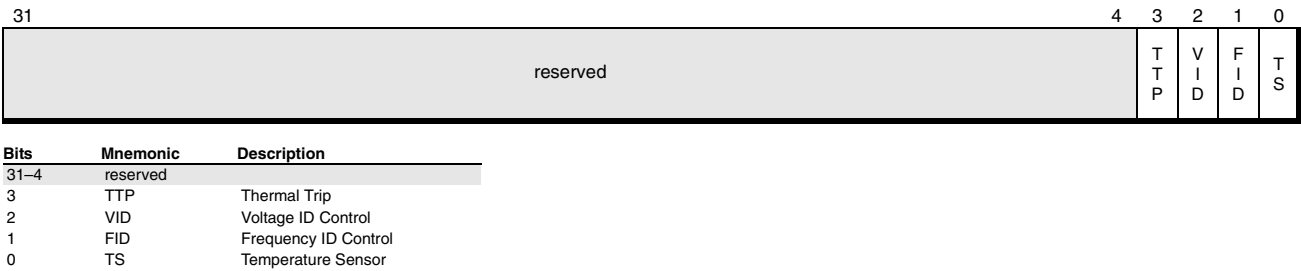


Figure 3-3. Advanced Power Management Features (EDX Register)

Function 8000_0008h: Address Sizes

Function 8000_0008h reports the maximum supported virtual-address and physical-address sizes.

EAX. Function 8000_0008h reports the address-size information in the EAX register. Table 3-14 on page 126 shows the format of the EAX register during execution of CPUID function 8000_0008h. The virtual-address and physical-address sizes that are returned indicate the address widths, in bits, supported by the processor

implementation. The values returned by this function are not influenced by enabling or disabling either long mode or physical-address extensions (CR4.PAE).

Table 3-14. CUID Long-Mode Address Sizes

| Register | Long-Mode Address Sizes | | |
|----------|-------------------------|----------------------|-----------------------|
| | Reserved | Virtual-Address Size | Physical-Address Size |
| EAX | Bits 31–16 | Bits 15–8 | Bits 7–0 |

EBX, ECX, and EDX. For function 8000_0008h, the EBX, ECX, and EDX registers are reserved.

Related Instructions

None

rFLAGS Affected

None

Exceptions

None

CQO**Convert Quadword to Double Quadword**

Converts a signed quadword in the RAX register into a signed double quadword in the RDX:RAX registers by copying the sign bit of the value in RAX into all of the bits of RDX. The value in RAX remains unchanged. This instruction helps avoid the overflow problem in signed number arithmetic.

This mnemonic is meaningful only in 64-bit mode. The CWD, CDQ, and CQD mnemonics share opcode 99. The operand-size attribute determines the size of the destination register.

| Mnemonic | Opcode | Description |
|----------|--------|---------------------------|
| CQO | 99 | Sign-extend RAX into RDX. |

Related Instructions

CWD, CDQ, CBW, CWDE, CDQE

rFLAGS Affected

None

Exceptions

None

CWD**Convert Word to Doubleword**

Converts a signed word in the AX register into a signed doubleword in the DX:AX registers by copying the sign bit of the value in AX into all of the bits of DX. The value in AX remains unchanged. This instruction helps avoid the overflow problem in signed number arithmetic.

The CWD, CDQ and CQO mnemonics share opcode 99. The operand-size attribute determines the size of the destination register.

| Mnemonic | Opcode | Description |
|----------|--------|-----------------------------|
| CWD | 99 | Sign-extends AX into DX:AX. |

Related Instructions

CDQ

rFLAGS Affected

None

Exceptions

None

CWDE**Convert Word to Doubleword**

Converts a signed word in the AX register into a signed doubleword in the EAX register by copying the sign bit of the value in AX into the higher 16 bits of EAX. The value in AX remains unchanged. This instruction helps avoid the overflow problem in signed number arithmetic.

In 64-bit mode, this instruction sign extends the value in the EAX register to the RAX register.

| Mnemonic | Opcode | Description |
|----------|--------|--------------------------|
| CWDE | 98 | Sign extend AX into EAX. |

Related Instructions

CDQ, CWD

rFLAGS Affected

None

Exceptions

None

DAA**Decimal Adjust after Addition**

Converts the value in the AL register into a packed BCD result and sets the CF and AF flags in the rFLAGS register if it detects a decimal carry.

Use this instruction after an ADD instruction that performs the binary addition of two 2-digit, packed BCD values and stores the result in the AL register.

The instruction performs the adjustment by adding 6 to the lower nibble of AL if that value is greater than 9 or if AF = 1 and adding 6 to the upper nibble of AL if that value is greater than 9 or if CF = 1.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| DAA | 27 | Decimal adjust AL. (Invalid in 64-bit mode.) |

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | U | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

DAS Decimal Adjust after Subtraction

Converts the value in the AL register into two packed BCD numbers and sets the CF and AF flags in the rFLAGS register if it detects a decimal borrow.

Use this instruction after a SUB instruction that performs a binary subtraction of one 2-digit, packed BCD value from another and stores the result in the AL register.

The instruction performs the adjustment by subtracting 6 from the lower nibble of AL if that value is greater than 9 or if AF = 1 and subtracting 6 from the upper nibble of AL if that value is greater than 9 or if CF = 1.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| DAS | 2F | Decimal adjusts AL after subtraction. (Invalid in 64-bit mode.) |

Related Instructions

DAA

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | U | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

DEC Decrement by 1

Subtracts 1 from the specified register or memory location. The CF flag is not affected.

The one-byte forms of this instruction (opcodes 48 through 4F) are used as REX prefixes in 64-bit mode. See “REX Prefixes” on page 14.

The memory-operand forms of the DEC instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.

| Mnemonic | Opcode | Description |
|----------------------|----------------|--|
| DEC <i>reg/mem8</i> | FE /1 | Decrement the contents of an 8-bit register or memory location by 1. |
| DEC <i>reg/mem16</i> | FF /1 | Decrement the contents of a 16-bit register or memory location by 1. |
| DEC <i>reg/mem32</i> | FF /1 | Decrement the contents of a 32-bit register or memory location by 1. |
| DEC <i>reg/mem64</i> | FF /1 | Decrement the contents of a 64-bit register or memory location by 1. |
| DEC <i>reg16</i> | 48 + <i>rw</i> | Decrement the contents of a 16-bit register by 1. (This opcode is are used as an REX prefix in 64-bit mode. See “REX Prefixes” on page 14.) |
| DEC <i>reg32</i> | 48 + <i>rd</i> | Decrement the contents of a 32-bit register by 1. (This opcode is used as an REX prefix in 64-bit mode. See “REX Prefixes” on page 14.) |

Related Instructions

INC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceed was data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

DIV Unsigned Divide

Divides the unsigned value in a register by the unsigned value in the specified register or memory location. The register to be divided depends on the size of the divisor.

When dividing a word, the instruction stores the quotient in the AL register and the remainder in the AH register.

When dividing a doubleword, quadword, or double quadword, the instruction assumes the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register.

The following table summarizes the action of this instruction:

| Operand Size | Dividend | Divisor | Quotient | Remainder | Maximum Quotient |
|--------------------------|----------|-----------|----------|-----------|------------------|
| Word/byte | AX | reg/mem8 | AL | AH | 255 |
| Doubleword/word | DX:AX | reg/mem16 | AX | DX | 65,535 |
| Quadword/doubleword | EDX:EAX | reg/mem32 | EAX | EDX | $2^{32} - 1$ |
| Double quadword/quadword | RDX:RAX | reg/mem64 | RAX | RDX | $2^{64} - 1$ |

The instruction truncates non-integral results towards 0 and the remainder is always less than the divisor. An overflow generates a #DE (divide error) exception, rather than setting the CF flag.

Division by zero generates a divide-by-zero exception.

| Mnemonic | Opcode | Description |
|----------------------|--------|---|
| DIV <i>reg/mem8</i> | F6 /6 | Perform unsigned division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH. |
| DIV <i>reg/mem16</i> | F7 /6 | Perform unsigned division of DX:AX by the contents of a 16-bit register or memory operand store the quotient in AX and the remainder in DX. |

| Mnemonic | Opcode | Description |
|----------------------|--------|---|
| DIV <i>reg/mem32</i> | F7 /6 | Perform unsigned division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX. |
| DIV <i>reg/mem64</i> | F7 /6 | Performs unsigned division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX. |

Related Instructions

IDIV, FDIV, FDIVP, FIDIV

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | U | | | | U | U | U | U | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:

Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Divide by zero, #DE | X | X | X | Source operand (divisor) was 0. |
| | X | X | X | The quotient was too large for the designated register. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

ENTER Create Procedure Stack Frame

Creates a stack frame for a procedure.

The first operand specifies the size of the stack frame and determines the registers for the current frame pointer (BP, EBP, or RBP) and the stack pointer (SP, ESP or RBP).

The second operand specifies the nesting level (0 to 31—the value is automatically masked to 5 bits). For nesting levels of 1 or greater, the processor creates additional stack frame pointers before adjusting the stack pointer. This action provides a called procedure with access points to other nested stack frames.

The `enter N, 0` (a nesting level of 0) instruction is equivalent to the following instruction sequence:

```
push    ebp        ; save current EBP
mov     ebp, esp    ; set stack frame pointer value
sub     esp, N      ; allocate space for local variables
```

The **ENTER** and **LEAVE** instructions provide support for block structured languages. The **LEAVE** instruction releases the stack frame on returning from a procedure.

In 64-bit mode, the operand size defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size.

| Mnemonic | Opcode | Description |
|--------------------------------|-----------------------|--|
| <code>ENTER imm16, 0</code> | <code>C8 iw 00</code> | Create a procedure stack frame. |
| <code>ENTER imm16, 1</code> | <code>C8 iw 01</code> | Create a nested stack frame for a procedure. |
| <code>ENTER imm16, imm8</code> | <code>C8 iw ib</code> | Create a nested stack frame for a procedure. |

Action

// See “Pseudocode Definitions” on page 46.

ENTER_START:

```
temp_ALLOC_SPACE = word-sized immediate specified in the instruction, zero-
extended to 64 bits
temp_PARAM_COUNT = byte-sized immediate specified in the instruction, zero-
extended to 64 bits
```

```
temp_PARAM_COUNT = temp_PARAM_COUNT AND 0x1f
// only keep 5 bits of param count
```

```

PUSH.v old_RBP

temp_RBP = RSP                // this value of RSP will eventually be loaded
                               // into RBP

IF (temp_PARAM_COUNT>0)      // push "temp_PARAM_COUNT" parameters to the
                               // stack
    FOR (I=1; I<temp_PARAM_COUNT; I++)
        // all but one of the parameters are copied
        // from higher up on the stack
    {
        temp_DATA = READ_MEM.v [SS:old_RBP-I*V]
        PUSH.v temp_DATA
    }
    PUSH.v temp_RBP           // the last parameter is the offset of the old
                               // value of RBP on the stack

RSP.s = RSP - temp_ALLOC_SPACE // leave "temp_ALLOC_SPACE" free bytes on
                               // the stack

temp_unused = READ_MEM.v [SS:RSP.s] // ENTER finishes with a memory check
                                     // on the final stack pointer

RBP.v = temp_RBP
EXIT

```

Related Instructions

LEAVE

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-----------------|------|-----------------|-----------|---|
| Stack, #SS | X | X | X | The EBP register pointed to a location that exceeded the limits of the current stack segment. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |

IDIV Signed Divide

Divides the signed value in a register by the signed value in the specified register or memory location. The register to be divided depends on the size of the divisor.

When dividing a word, the instruction stores the quotient in the AL register and the remainder in the AH register.

When dividing a doubleword, quadword, or double quadword, the instruction assumes the most-significant word of the dividend is in the rDX register and the least-significant word is in the rAX register. After the division, the instruction stores the quotient in the rAX register and the remainder in the rDX register.

The following table summarizes the action of this instruction:

| Operand Size | Dividend | Divisor | Quotient | Remainder | Quotient Range |
|--------------------------|----------|-----------|----------|-----------|---------------------------|
| Word/byte | AX | reg/mem8 | AL | AH | -128 to +127 |
| Doubleword/word | DX:AX | reg/mem16 | AX | DX | -32,768 to +32,767 |
| Quadword/doubleword | EDX:EAX | reg/mem32 | EAX | EDX | -2^{31} to $2^{32} - 1$ |
| Double quadword/quadword | RDX:RAX | reg/mem64 | RAX | RDX | -2^{63} to $2^{64} - 1$ |

The instruction truncates non-integral results towards 0. The sign of the remainder is always the same as the sign of the dividend, and the absolute value of the remainder is less than the absolute value of the divisor. An overflow generates a #DE (divide error) exception, rather than setting the OF flag.

To avoid overflow problems, precede this instruction with a CBW, CWDE, or CDQE instruction to sign-extend the dividend.

| Mnemonic | Opcode | Description |
|-----------------------|--------|--|
| IDIV <i>reg/mem8</i> | F6 /7 | Perform signed division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH. |
| IDIV <i>reg/mem16</i> | F7 /7 | Perform signed division of DX:AX by the contents of a 16-bit register or memory location and store the quotient in AX and the remainder in DX. |

| Mnemonic | Opcode | Description |
|-----------------------|--------|--|
| IDIV <i>reg/mem32</i> | F7 /7 | Perform signed division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX. |
| IDIV <i>reg/mem64</i> | F7 /7 | Perform signed division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX. |

Related Instructions

IMUL, CBW, CWDE, CDQE

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | U | | | | U | U | U | U | U |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|--------------|-----------|--|
| Divide by zero, #DE | X | X | X | The divisor operand was 0. |
| | X | X | X | The quotient was too large for the designated register. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory.. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

IMUL Signed Multiply

Multiplies two signed operands. The number of operands determines the form of the instruction.

If a single operand is specified, the instruction multiplies the value in the specified general-purpose register or memory location by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and stores the product in AX, DX:AX, EDX:EAX, or RDX:RAX, respectively.

If two operands are specified, the instruction multiplies the value in a general-purpose register (first operand) by an immediate value or the value in a general-purpose register or memory location (second operand) and stores the product in the first operand location.

If three operands are specified, the instruction multiplies the value in a general-purpose register or memory location (second operand), by an immediate value (third operand) and stores the product in a register (first operand).

The instruction sign-extends immediate value operands to the length of the destination format.

If the operation carries significant bits into the upper half of the destination location, the instruction sets the CF and OF flags to 1. If the result fits in the lower half of the destination location, it clears the CF and OF flags.

The two- and three-operand forms of the instruction truncate the result to the length of the destination location before storing it. Therefore, software should test the CF and OF flags after executing this instruction.

| Mnemonic | Opcode | Description |
|-----------------------|---------------|---|
| IMUL <i>reg/mem8</i> | F6 /5 | Multiply the contents of AL by the contents of an 8-bit memory or register operand and put the signed result in AX. |
| IMUL <i>reg/mem16</i> | F7 /5 | Multiply the contents of AX by the contents of a 16-bit memory or register operand and put the signed result in DX:AX. |
| IMUL <i>reg/mem32</i> | F7 /5 | Multiply the contents of EAX by the contents of a 32-bit memory or register operand and put the signed result in EDX:EAX. |
| IMUL <i>reg/mem64</i> | F7 /5 | Multiply the contents of RAX by the contents of a 64-bit memory or register operand and put the signed 128-bit result in RDX:RAX. |

| Mnemonic | Opcode | Description |
|-------------------------------------|---------------|---|
| IMUL <i>reg16, reg/mem16</i> | 0F AF /r | Multiply the contents of a 16-bit destination register by the contents of a 16-bit register or memory operand and put the signed result in the 16-bit destination register. |
| IMUL <i>reg32, reg/mem32</i> | 0F AF /r | Multiply the contents of a 32-bit destination register by the contents of a 32-bit register or memory operand and put the signed result in the 32-bit destination register. |
| IMUL <i>reg64, reg/mem64</i> | 0F AF /r | Multiply the contents of a 64-bit destination register by the contents of a 64-bit register or memory operand and put the signed result in the 64-bit destination register. |
| IMUL <i>reg16, reg/mem16, imm8</i> | 6B /r ib | Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 16-bit destination register. |
| IMUL <i>reg32, reg/mem32, imm8</i> | 6B /r ib | Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 32-bit destination register. |
| IMUL <i>reg64, reg/mem64, imm8</i> | 6B /r ib | Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 64-bit destination register. |
| IMUL <i>reg16, imm8</i> | 6B /r ib | Multiply the contents of a 16-bit register by a sign-extended immediate byte and put the signed result in the 16-bit destination register. |
| IMUL <i>reg32, imm8</i> | 6B /r ib | Multiply the contents of a 32-bit register by a sign-extended immediate byte and put the signed result in the 32-bit destination register. |
| IMUL <i>reg64, imm8</i> | 6B /r ib | Multiply the contents of a 64-bit register by a sign-extended immediate byte and put the signed result in the 64-bit destination register. |
| IMUL <i>reg16, reg/mem16, imm16</i> | 69 /r iw | Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate word and put the signed result in the 16-bit destination register. |
| IMUL <i>reg32, reg/mem32, imm32</i> | 69 /r id | Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate double and put the signed result in the 32-bit destination register. |
| IMUL <i>reg64, reg/mem64, imm32</i> | 69 /r id | Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate double and put the signed result in the 64-bit destination register. |
| IMUL <i>reg16, imm16</i> | 69 /r iw | Multiply the contents of a 16-bit destination register by a 16-bit immediate value and put the signed result in the destination register. |

| Mnemonic | Opcode | Description |
|--------------------------|------------------|---|
| IMUL <i>reg32, imm32</i> | 69 / <i>r id</i> | Multiply the contents of a 32-bit destination register by a 32-bit immediate value and put the signed result in the destination register. |
| IMUL <i>reg64, imm32</i> | 69 / <i>r id</i> | Multiply the contents of a 64-bit destination register by a 32-bit immediate value and put the signed result in the destination register. |

Related Instructions

IDIV

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|--------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

IN Input from Port

Transfers a byte, word, or doubleword from an I/O port (second operand) to the AL, AX or EAX register (first operand). The port address can be an 8-bit immediate value (00h to FFh) or contained in the DX register (0000h to FFFFh).

The port is in the processor's I/O address space. For 8-bit I/O port accesses, the opcode determines the port size. For 16-bit and 32-bit accesses, the operand-size attribute determines the port size.

If the CPL is higher than IOPL or running in virtual mode, check the IOPL bitmap.

| Mnemonic | Opcode | Description |
|---------------------|--------------|--|
| IN AL, <i>imm8</i> | E4 <i>ib</i> | Input a byte from the port at the address specified by <i>imm8</i> and put it into the AL register. |
| IN AX, <i>imm8</i> | E5 <i>ib</i> | Input a word from the port at the address specified by <i>imm8</i> and put it into the AX register. |
| IN EAX, <i>imm8</i> | E5 <i>ib</i> | Input a doubleword from the port at the address specified by <i>imm8</i> and put it into the EAX register. |
| IN AL, DX | EC | Input a byte from the port at the address specified by the DX register and put it into the AL register. |
| IN AX, DX | ED | Input a word from the port at the address specified by the DX register and put it into the AX register. |
| IN EAX, DX | ED | Input a doubleword from the port at the address specified by the DX register and put it into the EAX register. |

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| General protection, #GP | | X | X | At least one of the corresponding I/O permission bits was 1 in the TSS for the accessed I/O port. |

INC Increment by 1

Adds 1 to the specified register or memory location. The CF flag is not affected, even if the operand is incremented to 0000.

The one-byte forms of this instruction (opcodes 40 through 47) are used as REX prefixes in 64-bit mode. See “REX Prefixes” on page 14.

The memory-operand forms of the INC instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

To perform an increment operation that updates the CF flag, use an ADD instruction with an immediate operand of 1.

| Mnemonic | Opcode | Description |
|----------------------|----------------|--|
| INC <i>reg/mem8</i> | FE /0 | Increment the contents of an 8-bit register or memory location by 1. |
| INC <i>reg/mem16</i> | FF /0 | Increment the contents of a 16-bit register or memory location by 1. |
| INC <i>reg/mem32</i> | FF /0 | Increment the contents of a 32-bit register or memory location by 1. |
| INC <i>reg/mem64</i> | FF /0 | Increment the contents of a 64-bit register or memory location by 1. |
| INC <i>reg16</i> | 40 + <i>rw</i> | Increment the contents of a 16-bit register by 1. (These opcodes are used as REX prefixes in 64-bit mode. See “REX Prefixes” on page 14.) |
| INC <i>reg32</i> | 40 + <i>rd</i> | Increment the contents of a 32-bit register by 1. (These opcodes are used as REX prefixes in 64-bit mode. See “REX Prefixes” on page 14.) |

Related Instructions

ADD, DEC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| <i>Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.</i> | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory.. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

INS

Input String

INSB

INSW

INSD

Copies data from the I/O port specified in the DX register to an input buffer specified in the rDI register and increments or decrements the eDI register according to the setting of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rDI by 1, 2, or 4, depending on the number of bytes copied. If the DF flag is 1, it decrements the pointers by 1, 2, or 4.

In 16-bit and 32-bit mode, the INS instruction always uses ES as the data segment. In 64-bit mode, ES/GS prefixes are ignored, so INS always uses the unsegmented memory space.

The INS instructions use the explicit memory operand (first operand) to determine the size of the I/O port, but always use ES:[rDI] for the location of the input buffer. The explicit register operand (second operand) specifies the I/O port address and must always be DX.

The INSB, INSW, and INSD instructions copy byte, word, and doubleword data, respectively, from the I/O port (0000h to FFFFh) specified in the DX register to the input buffer specified in the ES:rDI registers.

If the operand size is 64-bits, the instruction behaves as if the operand size were 32-bits.

The INSx instructions use IOPL bitmap to verify access rights if in virtual x86 mode or if IOPL is less than CPL.

The INSx instructions support the REP prefix for block input of ECX bytes, words, or doublewords. For details about the REP prefix, see “Repeat Prefixes” on page 10.

| Mnemonic | Opcode | Description |
|-----------------------|---------------|---|
| INS <i>mem8</i> , DX | 6C | Input a byte from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |
| INS <i>mem16</i> , DX | 6D | Input a word from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |

| Mnemonic | Opcode | Description |
|-----------------------|---------------|---|
| INS <i>mem32</i> , DX | 6D | Input a doubleword from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |
| INSB | 6C | Input a byte from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |
| INSW | 6D | Input a word from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |
| INSD | 6D | Input a doubleword from the port at the address specified by the DX register and put it into the memory location specified in ES:rDI. |

Related Instructions

IN, OUT, OUTS, OUTSB, OUTSD, OUTSW

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|-------------|-------------------------|------------------|--|
| General protection, segment overrun, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| General protection, #GP | | X | X | The CPL was greater than the IOPL and one or more I/O permission bits were set in the TSS for the accessed port. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

INT Interrupt to Vector

Transfers execution to the interrupt or exception handler specified by an 8-bit unsigned immediate value. This value is an interrupt vector number (00h to FFh), which the processor uses as an index into the interrupt-descriptor table (IDT).

For detailed descriptions of the steps performed by `INTn` instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

See also the descriptions of the `INT 3` instruction on page 306 and the `INTO` instruction on page 155.

Note that if the interrupt causes a task switch, all `rFLAGS` may be modified.

| Mnemonic | Opcode | Description |
|-----------------------|--------------------|--|
| <code>INT imm8</code> | <code>CD ib</code> | Call interrupt service routine specified by interrupt vector <i>imm8</i> . |

Action

// See “Pseudocode Definitions” on page 46.

`INT_N_START:`

```
IF (REAL_MODE)
    INT_N_REAL
ELSIF (PROTECTED_MODE)
    INT_N_PROTECTED
ELSE // (VIRTUAL_MODE)
    INT_N_VIRTUAL
```

`INT_N_REAL:`

temp_int_n_vector = byte-sized interrupt vector specified in the instruction,
zero-extended to 64 bits

```
temp_RIP = READ_MEM.w [idt:temp_int_n_vector*4]
                // read target CS:RIP from the real-mode idt
temp_CS = READ_MEM.w [idt:temp_int_n_vector*4+2]
PUSH.w old_RFLAGS
PUSH.w old_CS
PUSH.w next_RIP
```



```

IF (temp_RIP>CS.limit)
    EXCEPTION [#GP]
CS.sel = temp_CS
CS.base = temp_CS SHL 4
RFLAGS.AC,TF,IF,RF cleared
RIP = temp_RIP
EXIT

```

INT_N_PROTECTED:

```

temp_int_n_vector = byte-sized interrupt vector specified in the instruction,
zero-extended to 64 bits
temp_idt_desc = READ_IDT (temp_int_n_vector)
IF (temp_idt_desc.attr.type = 'taskgate')
    TASK_SWITCH // using tss selector in the task gate as the target tss
IF (LONG_MODE) // the size of the gate controls the size of the stack pushes
    V=8-byte // long mode only uses 64-bit gates
ELIF ((temp_idt_desc.attr.type = 'intgate32')
    || (temp_idt_desc.attr.type = 'trapgate32'))
    V=4-byte // legacy mode, using a 32-bit gate
ELSE // gate is intgate16 or trapgate16
    V=2-byte // legacy mode, using a 16-bit gate

temp_RIP = temp_idt_desc.offset
IF (LONG_MODE)
    // in long mode, we need to read the 2nd half of a
    // 16-byte interrupt-gate from the gdt/ldt, to get the
    // upper 32 bits of the target RIP
{
    temp_upper = READ_MEM.q [idt:temp_int_n_vector*16+8]
    IF (temp_upper's extended attribute bits != 0)
        EXCEPTION [#GP(0)] // the extended attribute bits must be all zero
    temp_RIP = temp_RIP + (temp_upper SHL 32) // concatenate both halves of RIP
}

CS = READ_DESCRIPTOR (temp_idt_desc.segment, intcs_chk)

temp_CPL = CS.sel.rpl

IF (CPL=temp_CPL) // no privilege-level change
    IF (LONG_MODE)
        IF (temp_idt_desc.ist!=0)
            // in long mode, if the idt gate specifies an ist pointer,
            // a stack-switch is always done
            RSP = READ_MEM.q [tss:ist_index*8+28]

            RSP = RSP AND 0xFFFFFFFFFFFFFFF0
            // in long mode, interrupts/exceptions align rsp to a
            // 16-byte boundary

```

```

        PUSH.q old_SS      // in long mode, SS:RSP is always pushed to the stack
        PUSH.q old_RSP

    PUSH.v old_RFLAGS
    PUSH.v old_CS
    PUSH.v next_RIP

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
        EXCEPTION [#GP(0)]

    RFLAGS.VM,NT,TF,RF cleared
    RFLAGS.IF cleared if interrupt gate

    RIP = temp_RIP
    EXIT

ELSE // (CPL > temp_CPL), changing privilege level
{
    CPL = temp_CPL

    temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER
                           (CPL, temp_idt_desc.ist)

    IF (LONG_MODE)
        temp_RSP = temp_RSP AND 0xFFFFFFFFFFFFFFF0
                // in long mode, interrupts/exceptions align rsp
                // to a 16-byte boundary

    RSP.q = temp_RSP
    SS = temp_SS_desc

    PUSH.v old_SS      // #SS on the following pushes uses SS.sel as error code
    PUSH.v old_RSP
    PUSH.v old_RFLAGS
    PUSH.v old_CS
    PUSH.v next_RIP

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
        EXCEPTION [#GP(0)]

    RFLAGS.VM,NT,TF,RF cleared
    RFLAGS.IF cleared if interrupt gate
    RIP = temp_RIP
    EXIT
}

```

INT_N_VIRTUAL:

temp_int_n_vector = byte-sized interrupt vector specified in the instruction,
zero-extended to 64 bits

```
IF (CR4.VME=0)                // vme isn't enabled
    IF (RFLAGS.IOPL=3)
        INT_N_VIRTUAL_TO_PROTECTED
    ELSE
        EXCEPTION [#GP(0)]
```

```
temp_IRB_BASE = READ_MEM.w [tss:102]
                // check the vme Int-n Redirection Bitmap (IRB), to see
                // if we should redirect this interrupt to a virtual-mode
                // handler
temp_VME_REDUCTION_BIT = READ_BIT_ARRAY ([tss:temp_IRB_BASE-32],
temp_int_n_vector)
```

```
IF (temp_VME_REDUCTION_BIT=1)
    // the virtual-mode int-n bitmap bit is set, so don't
    // redirect this interrupt
```

```
IF (RFLAGS.IOPL=3)
    INT_N_VIRTUAL_TO_PROTECTED
ELSE
    EXCEPTION [#GP(0)]
```

```
ELSE                // redirect interrupt through virtual-mode idt
{
```

```
temp_RIP = READ_MEM.w [0:temp_int_n_vector*4]
            // read target CS:RIP from the virtual-mode idt at
            // linear address 0
temp_CS = READ_MEM.w [0:temp_int_n_vector*4+2]
```

```
IF (RFLAGS.IOPL < 3)
    old_RFLAGS = old_RFLAGS with VIF bit shifted into IF bit, and IOPL = 3
```

```
PUSH.w old_RFLAGS
PUSH.w old_CS
PUSH.w next_RIP
```

```
CS.sel = temp_CS
CS.base = temp_CS SHL 4
```

```
RFLAGS.TF,IF,RF cleared
RIP = temp_RIP        // RFLAGS.IF cleared if IOPL = 3
                     // RFLAGS.VIF cleared if IOPL < 3
```

```
EXIT
```

```
}
```

INT_N_VIRTUAL_TO_PROTECTED:

```

temp_idt_desc = READ_IDT (temp_int_n_vector)
IF (temp_idt_desc.attr.type = 'taskgate')
    TASK_SWITCH // using tss selector in the task gate as the target tss

IF ((temp_idt_desc.attr.type = 'intgate32')
    || (temp_idt_desc.attr.type = 'trapgate32'))
    // the size of the gate controls the size of the stack pushes
    V=4-byte // legacy mode, using a 32-bit gate
ELSE // gate is intgate16 or trapgate16
    V=2-byte // legacy mode, using a 16-bit gate

temp_RIP = temp_idt_desc.offset
CS = READ_DESCRIPTOR (temp_idt_desc.segment, intcs_chk)

IF (CS.sel.rpl!=0) // handler must run at cpl 0
    EXCEPTION [#GP(0)]

CPL = 0

temp_ist = 0 // legacy mode doesn't use ist pointers
temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER (CPL, temp_ist)

RSP.q = temp_RSP
SS = temp_SS_desc

PUSH.v old_GS // #SS on the following pushes use SS.sel as error code
PUSH.v old_FS
PUSH.v old_DS
PUSH.v old_ES
PUSH.v old_SS
PUSH.v old_RSP
PUSH.v old_RFLAGS // pushed with RF clear
PUSH.v old_CS
PUSH.v next_RIP

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]

DS = NULL // can't use virtual-mode selectors in protected mode
ES = NULL // can't use virtual-mode selectors in protected mode
FS = NULL // can't use virtual-mode selectors in protected mode
GS = NULL // can't use virtual-mode selectors in protected mode

RFLAGS.VM,NT,TF,RF cleared
RFLAGS.IF cleared if interrupt gate

RIP = temp_RIP
EXIT

```

Related Instructions

INT 3, INTO, BOUND

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|-----|-----|----|----|----|----|----|
| | | | | | 0 | 0 | | | | 1/0 | 1/0 | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| Invalid TSS, #TS (selector) | | X | X | The TSS stack-segment selector was null. |
| | | X | X | The new stack-segment selector in the TSS contained an RPL that is not equal to the DPL of the code-segment descriptor that was being accessed by the interrupt or trap gate. |
| | | X | X | The target stack-segment descriptor contained a DPL that is not equal to that of the code-segment descriptor being accessed by the interrupt or trap gate. |
| | | X | X | The TSS stack-segment was not a writable data segment. |
| | | X | X | The index in the stack-segment selector was beyond the descriptor table limits. |
| Segment not present, #NP (selector) | | X | X | One of the following was not present: code segment, TSS, trap gate, task gate, or interrupt gate. |
| Stack, #SS | X | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | X | | The stack segment limit was exceeded when a stack switch occurred. |
| | | | X | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| General protection, #GP | | X | X | The instruction pointer in the task gate, interrupt gate, trap gate, or IDT was outside the limits of the code segment. |
| | X | | | A memory address exceeded a data segment limit or was non-canonical. |
| | X | | | The exception vector was beyond the limits of the IDT. |
| | | X | | The IOPL is less than 3 and the DPL of the interrupt-, trap-, or task-gate descriptor was not equal to 3. |
| General protection, #GP (selector) | | X | X | The segment selector was null in the trap, interrupt, or task gate. |
| | | X | X | The selector index for the TSS, gate, or code segment, was beyond the limits of the descriptor table. |
| | | X | X | The exception vector was beyond the limits of the IDT. |
| | | X | X | The descriptor in the IDT was not for an interrupt, trap, or task gate. |
| | | X | X | An interrupt was generated by the INTn instruction and the DPL of an interrupt, trap, or task gate was less than the CPL. (For protected mode, also INT 3 or INTO instruction) |
| | | X | X | The segment selector in a trap gate or interrupt did not point to a code-segment descriptor. |
| | | X | X | The TI bit in the TSS segment selector indicated a local table. |
| Page fault, #PF | | | X | The segment descriptor for the TSS indicated that the TSS is not available or busy. (Protected mode only) |
| | | X | X | A page fault resulted from the execution of the instruction. |

INTO Interrupt to Overflow Vector

Checks the overflow flag (OF) in the rFLAGS register and calls the overflow exception (#OF) handler if the OF flag is set to 1. The INTO instruction detects overflow in signed number addition. In signed addition, OF is set to 1 if the magnitude of the positive or negative result is too large (overflow) or too small (underflow) to conform to its defined data type. See *AMD x86-64 Architecture Programmer's Manual Volume 1: Application Programming* for more information on the OF flag.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

For detailed descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| INTO | CE | Call overflow exception if the overflow flag is set. (Invalid in 64-bit mode.) |

Related Instructions

INT, INT 3, BOUND

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|-----|-----|----|----|----|----|----|
| | | | | | | | | | | 1/0 | 1/0 | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|--------------|-----------|---|
| Breakpoint, #BP | | X | | INT 3 instruction was executed. |
| Overflow, #OF | | X | | The INTO instruction was executed with OF set to 1. |
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| Invalid TSS, #TS (selector) | | X | X | The TSS stack-segment selector was null. |
| | | X | X | The new stack-segment selector in the TSS contained an RPL that was not equal to the DPL of the code-segment descriptor that was being accessed by the interrupt or trap gate. |
| | | X | X | The target stack-segment descriptor contained a DPL that was not equal to that of the code-segment descriptor being accessed by the interrupt or trap gate. |
| | | X | X | The TSS stack-segment was not a writable data segment. |
| | | X | X | The index in the stack-segment selector was beyond the descriptor table limits. |
| Segment not present, #NP (selector) | | X | X | One of the following was not present: code segment, TSS, trap gate, task gate, or interrupt gate. |
| Stack, #SS | X | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | X | | The stack segment limit was exceeded when a stack switch occurred. |
| | | | X | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |
| General protection, #GP | X | X | X | The instruction pointer in the in the task gate, interrupt gate, trap gate, or IDT was outside the limits of the code segment. |
| | | X | | The INT4 vector was beyond the limits of the IDT. The IOPL was less than 3 and the DPL of the interrupt-, trap-, or task-gate descriptor was not equal to 3. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| General protection, #GP (selector) | | X | X | The segment selector was null in the trap, interrupt, or task gate. |
| | | X | X | The selector index for the TSS, gate, or code segment, was beyond the limits of the descriptor table. |
| | | X | X | The exception vector was beyond the limits of the IDT. |
| | | X | X | The descriptor in the IDT was not for an interrupt, trap, or task gate. |
| | | X | X | The DPL of an interrupt, trap, or task gate was less than the CPL. |
| | | X | X | The segment selector in a trap gate or interrupt did not point to a code-segment descriptor. |
| | | X | X | The TI bit in the TSS segment selector indicated a local table. |
| | | X | X | The segment descriptor for the TSS indicated that the TSS is not available or busy. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |

Jcc Jump on Condition

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified by the condition code in the mnemonic (*cc*), jumps to the target instruction located at the specified relative offset. Otherwise, execution continues with the instruction following the Jcc instruction.

Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—*short and near conditional jumps*. Different opcodes correspond to different forms of one instruction. For example, the JO instruction (jump if overflow) has opcode 0Fh 80h for its near form and 70h for its short form, but the mnemonic is the same for both forms.

Mnemonics are provided to deal with the programming semantics of both signed and unsigned numbers. Instructions tagged A (above) and B (below) are intended to clarify unsigned integer code; those tagged G (greater) and L (less) are intended for use in signed integer manipulations.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit or 32-bit displacement value to 64 bits before adding the offset to the RIP. (The 32-bit displacement cannot be encoded because the operand size defaults to 64 bits.)

These instructions cannot perform far jumps (to other code segments). To create a far-conditional-jump code sequence corresponding to a high-level language statement like:

```
IF A = B THEN GOTO FarLabel
```

where FarLabel is located in another code segment, use the opposite condition in a conditional short jump before an unconditional far jump. Such a code sequence might look like:

```
cmp    A,B                ; compare operands
jne    NextInstr          ; continue program if not equal
jmp    far ptr WhenNE     ; far jump if operands are equal
```

```
NextInstr:                ; continue program
```

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

| Mnemonic | Opcode | Description |
|---|-----------------|--|
| JA <i>rel8off</i> JNB <i>rel8off</i> | 77 <i>cb</i> | Jump short if the carry flag (CF) and the zero flag (ZF) are both 0. |
| JA <i>rel16off</i> JNB <i>rel16off</i> | 0F 87 <i>cw</i> | Jump near if the carry flag (CF) and the zero flag (ZF) are both 0. |
| JA <i>rel32off</i> JNB <i>rel32off</i> | 0F 87 <i>cd</i> | Jump near if the carry flag (CF) and the zero flag (ZF) are both 0. |
| JAE <i>rel8off</i> JNB <i>rel8off</i> JNC <i>rel8off</i> | 73 <i>cb</i> | Jump short if the carry flag (CF) is 0. |
| JAE <i>rel16off</i> JNB <i>rel16off</i> JNC <i>rel16off</i> | 0F 83 <i>cw</i> | Jump near if the carry flag (CF) is 0. |
| JAE <i>rel32off</i> JNB <i>rel32off</i> JNC <i>rel32off</i> | 0F 83 <i>cd</i> | Jump near if the carry flag (CF) is 0. |
| JBE <i>rel8off</i> | 76 <i>cb</i> | Jump short if the carry flag (CF) is 1 or the zero flag (ZF) is 1. |
| JBE <i>rel16off</i> | 0F 86 <i>cw</i> | Jump near if the carry flag (CF) is 1 or the zero flag (ZF) is 1. |
| JBE <i>rel32off</i> | 0F 86 <i>cd</i> | Jump near if the carry flag (CF) is 1 or the zero flag (ZF) is 1. |
| JC <i>rel8off</i> JB <i>rel8off</i> JNAE <i>rel8off</i> | 72 <i>cb</i> | Jump short if the carry flag (CF) is 1. |
| JC <i>rel16off</i> JB <i>rel16off</i> JNAE <i>rel16off</i> | 0F 82 <i>cw</i> | Jump near if the carry flag (CF) is 1. |
| JC <i>rel32off</i> JB <i>rel32off</i> | 0F 82 <i>cd</i> | Jump near if the carry flag (CF) is 1. |
| JE <i>rel8off</i> JZ <i>rel8off</i> | 74 <i>cb</i> | Jump short if the zero flag (ZF) is 1. |
| JE <i>rel16off</i> JZ <i>rel16off</i> | 0F 84 <i>cw</i> | Jump near if the zero flag (ZF) is 1. |
| JE <i>rel32off</i> JZ <i>rel32off</i> | 0F 84 <i>cd</i> | Jump near if the zero flag (ZF) is 1. |
| JG <i>rel8off</i> JNLE <i>rel8off</i> | 7F <i>cb</i> | Jump short if the zero flag (ZF) is 0 and the sign flag (SF) is equal to the overflow flag (OF). |

| Mnemonic | Opcode | Description |
|--|-----------------|---|
| JG <i>rel16off</i> JNLE <i>rel16off</i> | 0F 8F <i>cw</i> | Jump near if the zero flag (ZF) is 0 and the sign flag (SF) is equal to the overflow flag (OF). |
| JG <i>rel32off</i> JNLE <i>rel32off</i> | 0F 8F <i>cd</i> | Jump near if the zero flag (ZF) is 0 and the sign flag (SF) is equal to the overflow flag (OF). |
| JGE <i>rel8off</i> JNL <i>rel8off</i> | 7D <i>cb</i> | Jump short if the sign flag (SF) and the overflow flag (OF) have the same value. |
| JGE <i>rel16off</i> JNL <i>rel16off</i> | 0F 8D <i>cw</i> | Jump near if the sign flag (SF) and the overflow flag (OF) have the same value. |
| JGE <i>rel32off</i> JNL <i>rel32off</i> | 0F 8D <i>cd</i> | Jump near if the sign flag (SF) and the overflow flag (OF) have the same value. |
| JL <i>rel8off</i> JNGE <i>rel8off</i> | 7C <i>cb</i> | Jump short if the sign flag (SF) is not equal to the overflow flag (OF). |
| JL <i>rel16off</i> JNGE <i>rel16off</i> | 0F 8C <i>cw</i> | Jump near if the sign flag (SF) is not equal to the overflow flag (OF). |
| JL <i>rel32off</i> JNGE <i>rel32off</i> | 0F 8C <i>cd</i> | Jump near if the sign flag (SF) is not equal to the overflow flag (OF). |
| JLE <i>rel8off</i> JNG <i>rel8off</i> | 7E <i>cb</i> | Jump short if the zero flag (ZF) is 1 and sign flag (SF) is not equal to the overflow flag (OF). |
| JLE <i>rel16off</i> JNG <i>rel16off</i> | 0F 8E <i>cw</i> | Jump near if the zero flag (ZF) is 1 and the sign flag (SF) is not equal to the overflow flag (OF). |
| JLE <i>rel32off</i> JNG <i>rel32off</i> | 0F 8E <i>cd</i> | Jump near if the zero flag (ZF) is 1 and the sign flag (SF) is not equal to the overflow flag (OF). |
| JNE <i>rel8off</i> JNZ <i>rel8off</i> | 75 <i>cb</i> | Jump short if the zero flag (ZF) is 0. |
| JNE <i>rel16off</i> JNZ <i>rel16off</i> | 0F 85 <i>cw</i> | Jump near if the zero flag (ZF) is 0. |
| JNE <i>rel32off</i> JNZ <i>rel32off</i> | 0F 85 <i>cd</i> | Jump near if the zero flag (ZF) is 0. |
| JNO <i>rel8off</i> | 71 <i>cb</i> | Jump short if the overflow flag (OF) is 0. |
| JNO <i>rel16off</i> | 0F 81 <i>cw</i> | Jump near if the overflow flag (OF) is 0. |
| JNO <i>rel32off</i> | 0F 81 <i>cd</i> | Jump near if the overflow flag (OF) is 0. |
| JNP <i>rel8off</i> JPO <i>rel8off</i> | 7B <i>cb</i> | Jump short if the parity flag (PF) is 0. |
| JNP <i>rel16off</i> JPO <i>rel16off</i> | 0F 8B <i>cw</i> | Jump near if the parity flag (PF) is 0. |

| Mnemonic | Opcode | Description |
|--|-----------------|--|
| JNP <i>rel32off</i> JPO <i>rel32off</i> | 0F 8B <i>cd</i> | Jump near if the parity flag (PF) is 0. |
| JO | 0F 80h | Jump near if the overflow flag (OF) is 1. |
| JO | 0F 70h | Jump short if the overflow flag (OF) is 1. |
| JS <i>rel8off</i> | 78 <i>cb</i> | Jump short if the sign flag (SF) is 1. |
| JS <i>rel16off</i> | 0F 88 <i>cw</i> | Jump near if the sign flag (SF) is 1. |
| JC <i>rel32off</i> | 0F 88 <i>cd</i> | Jump near if the carry flag (CF) is 1. |

Related Instructions

JCXZ, JECXZ, JMP, JRCXZ

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|-------------|-------------------------|------------------|---|
| General protection, #GP | X | X | | The target offset was beyond the limits of the CS segment or was outside of the effective address space of 0 to FFFFh. This condition can occur if a 32-bit address-size-override prefix is used. |
| | | | X | The target offset is beyond the limits of the CS segment. |

JCXZ Jump if CX Zero

JECXZ

JRCXZ

Checks the contents of the count register (rCX) and, if the value in rCX is 0, jumps to the target instruction located at the specified 8-bit relative offset. Otherwise, execution continues with the instruction following the JxCXZ instruction.

In 64-bit mode, the operand size defaults to 64 bits. The processor sign-extends the 8-bit displacement value to 64 bits before adding the offset to the RIP. It then masks the resulting RIP to the appropriate operand size (16 or 32 bits).

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

| Mnemonic | Opcode | Description |
|----------------------|--------------|--|
| JCXZ <i>rel8off</i> | E3 <i>cb</i> | Jump short if the count register (CX) is zero. |
| JECXZ <i>rel8off</i> | E3 <i>cb</i> | Jump short if the extended count register (ECX) is zero. |
| JRCXZ <i>rel8off</i> | E3 <i>cb</i> | Jump short if the 64-bit count register (RCX) is zero. |

Related Instructions

Jcc, JMP

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--|
| General protection, #GP | X | X | X | The target offset was beyond the limits of the CS segment. |

JMP

Jump

Transfers control unconditionally to a new address without saving current CS:RIP values. The difference between JMP and CALL is that the CALL instruction can return and continue execution with the instruction following the CALL, whereas JMP cannot return to the instruction following the JMP.

There are two types of jumps:

- When the target address is within the current code segment, the jump is called a *short jump* or *near jump*
- When the target address is outside the current code segment, the jump is called a *far jump*.

Within these two categories there are many ways to code the target address:

- *Relative Short Jump*—The target address must be within an 8-bit signed displacement of the rIP of the instruction after the JMP. This is a 2-byte instruction. The first byte is opcode EBh and the second byte is the signed-number displacement, which is added to the rIP of the instruction following the JMP to get the target address.
- *Relative Near Jump*—The target address must be within a 16-bit or 32-bit signed displacement of the rIP of the instruction after the JMP. This is a three-byte instruction. The first byte is opcode E9h and the next two bytes are the signed-number displacement. The displacement is added to the rIP of the instruction following the JMP to get the target address.
- *Register-Indirect Near Jump*—The target address is contained in a register. Any non-segment register can be used to specify the target address.
- *Memory-Indirect Near Jump*—The target address is in a memory location whose address is pointed to by a register.
- *Direct Far Jump*—The target address is outside the current code segment. Therefore, software must specify both the code segment and the offset of the target address. This is a five-byte instruction: the opcode EAh and 4 bytes for the segment and offset of the target address.
- *Memory-Indirect Far Jump*—The target address (both CS:rIP) is in a memory location whose address is pointed to by a register.

For near jumps in 64-bit mode, the operand size defaults to 64 bits. For the EB opcode, the resulting RIP = starting RIP + 8-bit displacement sign-extended to 64 bits. For the E9 opcode, the resulting RIP = starting RIP + 32-bit displacement sign-extended to 64 bits (the 16-bit displacement cannot be encoded because the operand size defaults to

64 bits). For the FF /4 opcode, the resulting RIP = the 64-bit offset from the register or memory.

For far jumps in 64-bit mode, the EA opcode is invalid, and the FF /5 opcode does the following:

- For 32-bit operand size, if the selector points to a gate, then RIP = zero-extended 32-bit offset from the gate, else RIP = zero-extended 32-bit offset from the far pointer referenced in instruction.
- For 64-bit operand size, if the selector points to a gate, then RIP = 64-bit offset from the gate, else RIP = zero-extended 32-bit offset from the far pointer referenced in instruction.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

| Mnemonic | Opcode | Description |
|----------------------|---------------|--|
| <i>JMP rel8off</i> | EB <i>cb</i> | Short relative jump. |
| <i>JMP rel16off</i> | E9 <i>cw</i> | Near relative jump. (Cannot be encoded in 64-bit mode.) |
| <i>JMP rel32off</i> | E9 <i>cd</i> | Near relative jump. |
| <i>JMP reg/mem16</i> | FF /4 | Absolute indirect near jump to instruction located in the specified register or memory operand. |
| <i>JMP reg/mem32</i> | FF /4 | Absolute indirect near jump to instruction located in the specified register or memory operand. |
| <i>JMP reg/mem64</i> | FF /4 | Absolute indirect near jump to instruction located at 64-bit offset from the specified register or memory operand. |
| <i>JMP pntr16:16</i> | EA <i>cd</i> | Absolute far jump to the address specified by the pointer. (Invalid in 64-bit mode.) |
| <i>JMP pntr16:32</i> | EA <i>cp</i> | Absolute far jump to the address specified by the pointer. (Invalid in 64-bit mode.) |
| <i>JMP mem16:16</i> | FF /5 | Absolute indirect far jump to the address given in the memory operand. |
| <i>JMP mem16:32</i> | FF /5 | Absolute indirect far jump to the address given in the memory operand. |

Action

// Far jumps (JMPF)
// See “Pseudocode Definitions” on page 46.

JMPF_START:

```
IF (REAL_MODE)
    JMPF_REAL_OR_VIRTUAL
ELIF (PROTECTED_MODE)
    JMPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    JMPF_REAL_OR_VIRTUAL
```

JMPF_REAL_OR_VIRTUAL:

```
IF (OPCODE = jmpf [mem])
{
    temp_RIP = READ_MEM.z [mem]
    temp_CS  = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE = jmpf direct)
{
    temp_RIP = z-sized offset specified in the instruction,
               zero-extended to 64 bits
    temp_CS  = selector specified in the instruction
}

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]

CS.sel = temp_CS
CS.base = temp_CS SHL 4
RIP = temp_RIP
EXIT
```

JMPF_PROTECTED:

```
IF (OPCODE = jmpf [mem])
{
    temp_offset = READ_MEM.z [mem]
    temp_sel    = READ_MEM.w [mem+Z]
}
ELSE // (OPCODE = jmpf direct)
{
    IF (64BIT_MODE)
        EXCEPTION [#UD] // 'jmpf direct' is illegal in 64-bit mode
    temp_offset = z-sized offset specified in the instruction,
                   zero-extended to 64 bits
    temp_sel    = selector specified in the instruction
}
temp_desc = READ_DESCRIPTOR (temp_sel, cs_chk)
           // read descriptor, perform protection and type checks
IF (temp_desc.attr.type = 'available_tss')
```

```

        TASK_SWITCH      // using temp_sel as the target tss selector
    ELSIF (temp_desc.attr.type = 'taskgate')
        TASK_SWITCH      // using the tss selector in the task gate as the
                        // target tss
    ELSIF (temp_desc.attr.type = 'code')
        // if the selector refers to a code descriptor, then
        // the offset we read is the target RIP
    {
        temp_RIP = temp_offset
        CS = temp_desc
        IF ((!64BIT_MODE) && (temp_RIP > CS.limit))
            // temp_RIP can't be non-canonical because
            // it's a 16- or 32-bit number, zero-extended to 64 bits
        {
            EXCEPTION [#GP(0)]
        }
        RIP = temp_RIP
        EXIT
    }
    ELSE
    {
        // (temp_desc.attr.type = 'callgate')
        // if the selector refers to a call gate, then
        // the target CS and RIP both come from the call gate
        temp_RIP = temp_desc.offset

        IF (LONG_MODE)
        {
            // in long mode, we need to read the 2nd half of a 16-byte call-gate
            // from the gdt/ldt to get the upper 32 bits of the target RIP
            temp_upper = READ_MEM.q [temp_sel+8]
            IF (temp_upper [44:40] != 0)
                EXCEPTION [#GP(0)] // make sure the extended attribute bits
                                // are all zero
            temp_RIP = temp_RIP + (temp_upper SHL 32)
            // concatenate both halves of RIP
        }
        CS = READ_DESCRIPTOR (temp_desc.segment, clg_chk)
            // set up new CS base, attr, limits
        IF ((64BIT_MODE) && (temp_RIP is non-canonical)
            || (!64BIT_MODE) && (temp_RIP > CS.limit))
            EXCEPTION [#GP(0)]
        RIP = temp_RIP
        EXIT
    }
}

```

Related Instructions

Jcc, JCXZ, JECXZ, JRCXZ

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | The absolute far JMP opcode (EAh) of this instruction was not recognized in 64-bit mode. |
| Segment not present, #NP (selector) | | | X | The accessed code segment, call gate, task gate, or TSS was not present. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The offset in the TSS, call gate, or target operand exceeded the limits of the code segment. |
| | | | X | There was a null segment selector in the destination operand, TSS, call gate, or task gate. |
| | | | X | A null data segment was being used to reference memory. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| General protection, #GP (selector) | | | X | The segment selector index was outside the descriptor table limits. |
| | | | X | The segment descriptor to which the segment selector points in the destination operand was not for a conforming-code segment, non-conforming-code segment, call gate, task gate, or task state segment. |
| | | | X | The DPL for a nonconforming-code segment was not equal to the CPL. |
| | | | X | When not using a call gate, the RPL for the segment 's segment selector was greater than the CPL. |
| | | | X | A conforming-code segment's DPL was greater than the CPL. |
| | | | X | The DPL from a call-gate, task-gate, or TSS segment descriptor was less than the CPL, or than the RPL of the call-gate, task-gate, or TSS segment selector. |
| | | | X | The segment descriptor for selector in a call gate did not indicate it was a code segment. |
| | | | X | The segment descriptor for the segment selector in the task gate did not indicate available TSS. |
| | | | X | The TI bit in the TSS segment selector indicated a local table. |
| | | | X | The segment descriptor for the TSS indicated that the TSS is not available or busy. |
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or is non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LAHF**Load Status Flags into AH Register**

Loads the lower 8 bits of the rFLAGS register, including the trap flag (TR), sign flag (SF), zero flag (ZF), auxiliary carry flag (AF), parity flag (PF), and carry flag (CF), into the AH register.

The instruction sets the reserved bits 1, 3, and 5 of the rFLAGS register to 1, 0, and 0, respectively, in the AH register.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| LAHF | 9F | Load the SF, ZF, AF, PF, and CF flags into the AH register. (Invalid in 64-bit mode.) |

Related Instructions

SAHF

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

LDS Load Far Pointer

LES

LFS

LGS

LSS

Loads a far pointer from a memory location (second operand) into a segment register (mnemonic) and general-purpose register (first operand). The instruction stores the 16-bit segment selector of the pointer into the segment register and the 16-bit or 32-bit offset portion into the general-purpose register. The operand-size attribute determines whether the pointer is 32-bit or 48-bit.

In protected mode, the instruction loads associated segment-descriptor information into the hidden portion of the specified segment register.

It can load a null segment selector (0000h through 0003h) without raising a #GP exception. However, a subsequent reference to such a segment causes a #GP exception and no memory reference to the segment occurs.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|---|
| <i>LDS reg16, mem16:16</i> | <i>C5 /r</i> | Load DS:reg16 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LDS reg32, mem16:32</i> | <i>C5 /r</i> | Load DS:reg32 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LES reg16, mem16:16</i> | <i>C4 /r</i> | Load ES:reg16 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LES reg32, mem16:32</i> | <i>C4 /r</i> | Load ES:reg32 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LFS reg16, mem16:16</i> | <i>0F B4 /r</i> | Load FS:reg16 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LFS reg32, mem16:32</i> | <i>0F B4 /r</i> | Load FS:reg32 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| <i>LGS reg16, mem16:16</i> | <i>0F B5 /r</i> | Load GS:reg16 with a specified far pointer from memory. (Invalid in 64-bit mode.) |

| Mnemonic | Opcode | Description |
|----------------------------|----------|---|
| LGS <i>reg32, mem16:32</i> | 0F B5 /r | Load GS:reg32 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| LSS <i>reg16, mem16:16</i> | 0F B2 /r | Load SS:reg16 with a specified far pointer from memory. (Invalid in 64-bit mode.) |
| LSS <i>reg32, mem16:32</i> | 0F B2 /r | Load SS:reg32 with a specified far pointer from memory. (Invalid in 64-bit mode.) |

Related Instructions

LEA, LES, LFS, LGS, LSS, STOS, STOSB, STOSW, STOSD

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The source operand was a register. |
| | | | X | Software was executing in 64-bit mode. |
| Segment not present, #NP (selector) | | | X | The DS, ES, FS, or GS register was being loaded with a non-null segment selector and the segment was marked not present. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | | X | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The null selector was loaded into the SS register. |
| | | | X | A null data segment was being used to reference memory. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| General protection, #GP (selector) | | | X | The SS register was being loaded and any of the following was true: the segment selector index was not within the descriptor table limits, the segment selector RPL was not equal to CPL, the segment was a non-writable data segment, or DPL was not equal to CPL. |
| | | | X | The DS, ES, FS, or GS register was being loaded with a non-null segment selector and any of the following was true: the segment selector index was not within descriptor table limits, the segment was neither a data nor a readable code segment, or the segment was a data or nonconforming-code segment and both RPL and CPL are greater than DPL. |
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LEA

Computes the effective address of a memory location (second operand) and stores it in a general-purpose register (first operand).

The address size of the memory location and the size of the register determine the specific action taken by the instruction, as follows:

- If the address size and the register size are the same, the instruction stores the effective address as computed.
- If the address size is longer than the register size, the instruction truncates the effective address to the size of the register.
- If the address size is shorter than the register size, the instruction zero-extends the effective address to the size of the register.

If the second operand is not a memory location (i.e., it is a register), an undefined-opcode exception occurs.

The LEA instruction is related to the MOV instruction, which copies data from a memory location to a register, but LEA takes the address of the source operand, whereas MOV takes the contents of the memory location specified by the source operand. In the simplest cases, LEA can be replaced with MOV. For example:

```
lea  eax, [ebx]
```

has the same effect as:

```
mov eax, ebx
```

However, LEA allows software to use any valid addressing mode for the source operand. For example:

```
lea  eax, [ebx+edi]
```

loads the sum of the EBX and EDI registers into the EAX register. This could not be accomplished by a single MOV instruction.

The LEA instruction has a limited capability to perform multiplication of operands in general-purpose registers using scaled-index addressing. For example:

```
lea eax, [ebx+ebx*8]
```

loads the value of the EBX register, multiplied by 9, into the EAX register. Possible values of multipliers are 2, 4, 8, 3, 5, and 9.

The LEA instruction is widely used in string-processing and array-processing to initialize an index register (rSI or rDI) before performing string instructions such as MOVSx. It is also used to initialize the rBX register before performing the XLAT instruction in programs that perform character translations. In data structures, the LEA instruction can calculate addresses of operands stored in memory, and in particular, addresses of array or string elements.

| Mnemonic | Opcode | Description |
|-------------------------|--------|---|
| LEA <i>reg16, mem16</i> | 8D /r | Store effective <i>mem16:32</i> address in a 16-bit register. |
| LEA <i>reg32, mem32</i> | 8D /r | Store effective <i>mem16:32</i> address in a 32-bit register. |
| LEA <i>reg64, mem64</i> | 8D /r | Store effective <i>mem16:64</i> address in a 64-bit register. |

Related Instructions

LDS, LES, LFS, LGS, LSS, STOS, STOSB, STOSW, STOSD

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|------------------------------------|
| Invalid opcode, #UD | X | X | X | The source operand was a register. |

LEAVE**Delete Procedure Stack Frame**

Releases a stack frame created by a previous ENTER instruction. To release the frame, it copies the frame pointer (in the rBP register) to the stack pointer register (rSP), and then pops the old frame pointer from the stack into the rBP register, thus restoring the stack frame of the calling procedure.

The LEAVE instruction is equivalent to the following operations:

```
MOV SP,BP          ;MOV ESP,EBP in 32-bit mode, MOV RSP,RBP in 64-bit mode
POP BP             ;POP EBP in 32-bit mode, POP RBP in 64-bit mode
```

To return program control to the calling procedure, execute a RET instruction after the LEAVE instruction.

In 64-bit mode, the operand size defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size.

| Mnemonic | Opcode | Description |
|-----------------|---------------|---|
| LEAVE | C9 | Set the stack pointer register SP to the value in the BP register and pop BP. |
| LEAVE | C9 | Set the stack pointer register ESP to the value in the EBP register and pop EBP. (No prefix for encoding this in 64-bit mode.) |
| LEAVE | C9 | Set the stack pointer register RSP to the value in the RBP register and pop RBP. |

Related Instructions

ENTER

rFLAGS Affected

None

LFENCE

Load Fence

Acts as a barrier to force strong memory ordering (serialization) between load instructions preceding the LFENCE and load instructions that follow the LFENCE. A weakly-ordered memory system allows hardware to reorder reads and writes between the processor and memory. The LFENCE instruction guarantees that the system completes all previous loads before executing subsequent loads.

The LFENCE instruction is weakly-ordered with respect to store instructions, data and instruction prefetches, and the SFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an LFENCE.

In addition to load instructions, the LFENCE instruction is strongly ordered with respect to other LFENCE instructions, MFENCE instructions, and serializing instructions.

Support for the LFENCE instruction is indicated when the SSE2 bit (bit 26) is set to 1 in EDX after executing CPUID with extended function 8000_0001h.

| Mnemonic | Opcode | Description |
|----------|----------|---|
| LFENCE | 0F AE E8 | Force strong ordering of (serialize) load operations. |

Related Instructions

MFENCE, SFENCE

rFLAGS Affected

None

Exceptions

None

LODS

Load String

LODSB

LODSW

LODSD

LODSQ

Copies the byte, word, doubleword, or quadword in the memory location pointed to by the DS:rSI registers to the AL, AX, EAX, or RAX register, depending on the size of the operand, and increments or decrements the rSI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments rSI; otherwise, it decrements rSI. It increments or decrements rSI by 1, 2, 4, or 8, depending on the number of bytes being loaded.

The forms of the LODS instruction with an explicit operand address the operand at *seg*:*[rSI]*. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix. The explicit operand serves only to specify the type (size) of the value being copied and the specific registers used.

The no-operands forms of the instruction always use the DS:*[rSI]* registers to point to the value to be copied (they do not allow a segment prefix). The mnemonic determines the size of the operand and the specific registers used.

The LODSx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 10. More often, software uses the LODSx instruction inside a loop controlled by a LOOPcc instruction as a more efficient replacement for instructions like:

```
mov eax, dword ptr ds:[esi]
add esi, 4
```

The LODSQ instruction is meaningful only in 64-bit mode.

| Mnemonic | Opcode | Description |
|-------------------|---------------|------------------------------------|
| LODS <i>mem8</i> | AC | Load byte at DS:SI into AL |
| LODS <i>mem16</i> | AD | Load word at DS:SI into AX. |
| LODS <i>mem32</i> | AD | Load doubleword at DS:ESI into EAX |
| LODS <i>mem64</i> | AD | Load quadword at DS:RSI into RAX |

| Mnemonic | Opcode | Description |
|-----------------|---------------|------------------------------------|
| LODSB | AC | Load byte at DS:SI into AL |
| LODSW | AD | Load the word at DS:SI into AX |
| LODSD | AD | Load doubleword at DS:ESI into EAX |
| LODSQ | AD | Load quadword at DS:RSI into RAX |

Related Instructions

LODSB, LODSD, LODSQ, LODSW

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LOOP Loop

LOOPE

LOOPNE

LOOPNZ

LOOPZ

Decrements the count register (rCX) by 1, then, if rCX is not 0 and the ZF flag meets the condition specified by the mnemonic, it jumps to the target instruction specified by the signed 8-bit relative offset. Otherwise, it continues with the next instruction after the LOOPcc instruction. This instruction is equivalent to the following instructions:

```
DEC  CX
JNZ  target
```

Assembly code typically specifies the offset as a label, but at the machine code level, it is encoded as a signed, 8-bit immediate value that is added to the instruction pointer. The offset must be in the range –128 to +127. The jump location specified by the offset is the beginning of the loop.

The LOOP instruction ignores the state of the ZF flag.

The LOOPE and LOOPZ instructions jump if rCX is not 0 and the ZF flag is set to 1. In other words, the instruction exits the loop (falls through to the next instruction) if rCX becomes 0 or ZF = 0.

The LOOPNE and LOOPNZ instructions jump if rCX is not 0 and ZF flag is cleared to 0. In other words, the instruction exits the loop if rCX becomes 0 or ZF = 1.

The LOOPcc instruction does not change the state of the ZF flag. Typically, the loop contains a compare instruction to set or clear the ZF flag.

In 64-bit mode, the operand size defaults to 64 bits without the need for a REX prefix, and the processor sign-extends the 8-bit offset before adding it to the RIP.

| Mnemonic | Opcode | Description |
|----------------------|--------------|---|
| LOOP <i>rel8off</i> | E2 <i>cb</i> | Jump short if rCX is not zero after decrementing count. |
| LOOPE <i>rel8off</i> | E1 <i>cb</i> | Jump short if rCX is not zero after decrementing count and ZF is 1. |

| Mnemonic | Opcode | Description |
|-----------------------|--------------|---|
| LOOPNE <i>rel8off</i> | E0 <i>cb</i> | Jump short if rCX is not zero after decrementing count and ZF is 0. |
| LOOPNZ <i>rel8off</i> | E0 <i>cb</i> | Jump short if rCX is not zero after decrementing count and ZF is 0. |
| LOOPZ <i>rel8off</i> | E1 <i>cb</i> | Jump short if rCX is not zero after decrementing count and ZF is 1. |

Related Instructions

LOOPE, LOOPZ, LOOPNE, LOOPNZ

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--|
| General protection, #GP | | | X | The target offset was beyond the limits of the code segment. |

MFENCE

Memory Fence

Acts as a barrier to force strong memory ordering (serialization) between load and store instructions preceding the MFENCE, and load and store instructions that follow the MFENCE. A weakly-ordered memory system allows the hardware to reorder reads and writes between the processor and memory. The MFENCE instruction guarantees that the system completes all previous memory accesses before executing subsequent accesses.

The MFENCE instruction is weakly-ordered with respect to data and instruction prefetches. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an MFENCE.

In addition to load and store instructions, the MFENCE instruction is strongly ordered with respect to other MFENCE instructions, LFENCE instructions, SFENCE instructions, serializing instructions, and CLFLUSH instructions.

Support for the MFENCE instruction is indicated when the SSE2 bit (bit 26) is set to 1 in EDX after executing CPUID with extended function 8000_0001h.

| Mnemonic | Opcode | Description |
|----------|----------|--|
| MFENCE | OF AE F0 | Force strong ordering of (serialized) load and store operations. |

Related Instructions

LFENCE, SFENCE

rFLAGS Affected

None

Exceptions

None

MOV**Move**

Copies an immediate value or the value in a general-purpose register, segment register, or memory location (second operand) to a general-purpose register, segment register, or memory location. The source and destination must be the same size (byte, word, doubleword, or quadword) and cannot both be memory locations.

In opcodes A0 through A3, the memory offsets (called *moffsets*) are address sized. In 64-bit mode, memory offsets default to 64 bits. The 64-bit memory offset does not need a REX prefix. The B8 through BF (B8 +*rq*) opcodes, in 64-bit mode, are the only cases that support a 64-bit immediate value (in all other cases, immediate values are a maximum of 32 bits in 64-bit mode).

For segment-register moves in legacy mode with a 32-bit operand size, the processor zero-extends 16-bit register results to 32 bits. For segment-register moves in 64-bit mode with 32-bit and 64-bit operand sizes, the processor zero-extends 32-bit register results to 64 bits. If the destination operand specifies a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector.

When the MOV instruction is used to load the SS register, the processor blocks all interrupts until after the execution of the following instruction. This action allows the following instruction to be a MOV instruction to load a stack pointer into the ESP register before an interrupt occurs (MOV ESP, val). However, the LSS instruction provides a more efficient method of loading SS and ESP.

Attempting to use the MOV instruction to load the CS register generates an invalid opcode exception (#UD). Use the far JMP, CALL, or RET instructions to load the CS register.

To initialize a register to 0, rather than using a MOV instruction, it may be more efficient to use the XOR instruction with identical destination and source operands.

| Mnemonic | Opcode | Description |
|-----------------------------|--------------|--|
| MOV <i>reg/mem8, reg8</i> | 88/ <i>r</i> | Move the contents of an 8-bit register to an 8-bit destination register or memory operand. |
| MOV <i>reg/mem16, reg16</i> | 89/ <i>r</i> | Move the contents of a 16-bit register to a 16-bit destination register or memory operand. |
| MOV <i>reg/mem32, reg32</i> | 89/ <i>r</i> | Move the contents of a 32-bit register to a 32-bit destination register or memory operand. |

| Mnemonic | Opcode | Description |
|--------------------------------------|----------------|--|
| MOV <i>reg/mem64, reg64</i> | 89 / <i>r</i> | Move the contents of a 64-bit register to a 64-bit destination register or memory operand. |
| MOV <i>reg8, reg/mem8</i> | 8A / <i>r</i> | Move the contents of an 8-bit register or memory operand to an 8-bit destination register. |
| MOV <i>reg16, reg/mem16</i> | 8B / <i>r</i> | Move the contents of a 16-bit register or memory operand to a 16-bit destination register. |
| MOV <i>reg32, reg/mem32</i> | 8B / <i>r</i> | Move the contents of a 32-bit register or memory operand to a 32-bit destination register. |
| MOV <i>reg64, reg/mem64</i> | 8B / <i>r</i> | Move the contents of a 64-bit register or memory operand to a 64-bit destination register. |
| MOV <i>reg16/32/64/mem16, segReg</i> | 8C / <i>r</i> | Move the contents of a segment register to a 16-bit, 32-bit, or 64-bit destination register or to a 16-bit memory operand. |
| MOV <i>segReg, reg16/32/64/mem16</i> | 8E / <i>r</i> | Move the low 16-bits of a 16-bit, 32-bit, or 64-bit register or of a 16-bit memory operand to a segment register. |
| MOV AL, <i>offset8</i> | A0 | Move 8-bit data at a specified memory offset to the AL register. |
| MOV AX, <i>offset16</i> | A1 | Move 16-bit data at a specified memory offset to the AX register. |
| MOV EAX, <i>offset32</i> | A1 | Move 32-bit data at a specified memory offset to the EAX register. |
| MOV RAX, <i>offset64</i> | A1 | Move 64-bit data at a specified memory offset to the RAX register. |
| MOV <i>offset8, AL</i> | A2 | Move the contents of the AL register to an 8-bit memory offset. |
| MOV <i>offset16, AX</i> | A3 | Move the contents of the AX register to a 16-bit memory offset. |
| MOV <i>offset32, EAX</i> | A3 | Move the contents of the EAX register to a 32-bit memory offset. |
| MOV <i>offset64, RAX</i> | A3 | Move the contents of the RAX register to a 64-bit memory offset. |
| MOV <i>reg8, imm8</i> | B0 + <i>rb</i> | Move an 8-bit immediate value into an 8-bit register. |
| MOV <i>reg16, imm16</i> | B8 + <i>rw</i> | Move a 16-bit immediate value into a 16-bit register. |
| MOV <i>reg32, imm32</i> | B8 + <i>rd</i> | Move an 32-bit immediate value into a 32-bit register. |
| MOV <i>reg64, imm64</i> | B8 + <i>rq</i> | Move an 64-bit immediate value into a 64-bit register. |
| MOV <i>reg/mem8, imm8</i> | C6 /0 | Move an 8-bit immediate value to an 8-bit register or memory operand. |
| MOV <i>reg/mem16, imm16</i> | C7 /0 | Move a 16-bit immediate value to a 16-bit register or memory operand. |

| Mnemonic | Opcode | Description |
|-----------------------------|---------------|---|
| MOV <i>reg/mem32, imm32</i> | C7 /0 | Move a 32-bit immediate value to a 32-bit register or memory operand. |
| MOV <i>reg/mem64, imm32</i> | C7 /0 | Move a 32-bit immediate value to a 64-bit register or memory operand. |

Related InstructionsMOV(CR*n*), MOV(DR*n*), MOVD, MOV_S, MOV_{SB}, MOV_{SD}, MOV_{SW}, MOV_{SX}, MOV_{ZX}**rFLAGS Affected**

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|-------------|-------------------------|------------------|--|
| Invalid opcode, #UD | X | X | X | An attempt was made to load the CS register. |
| Segment not present, #NP (selector) | | | X | The DS, ES, FS, or GS register was being loaded with a non-null segment selector and the segment was marked not present. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | | X | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The null selector was loaded into the SS register. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| General protection, #GP (selector) | | | X | The segment selector index was outside the descriptor table limits. |
| | | | X | The SS register was being loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL. |
| | | | X | The SS register was being loaded and the segment pointed to was a non-writable data segment. |
| | | | X | The DS, ES, FS, or GS register was being loaded and the segment pointed to was not a data or readable code segment. |
| | | | X | The DS, ES, FS, or GS register was being loaded and the segment pointed to was a data or nonconforming code segment, but both the RPL and the CPL were greater than the DPL. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MOVD**Move Doubleword or Quadword**

The MOVD instruction moves a 32-bit or 64-bit value in one of the following ways:

- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 or 64 bits of an XMM register, with zero-extension to 128 bits
- from the low-order 32 or 64 bits of an XMM to a 32-bit or 64-bit general-purpose register or memory location
- from a 32-bit or 64-bit general-purpose register or memory location to the low-order 32 bits (with zero-extension to 64 bits) or the full 64 bits of an MMX register
- from the low-order 32 or the full 64 bits of an MMX register to a 32-bit or 64-bit general-purpose register or memory location

| Mnemonic | Opcode | Description |
|----------------------------|---------------|---|
| MOVD <i>xmm, reg/mem32</i> | 66 0F 6E/r | Move 32-bit value from a general-purpose register or 32-bit memory location to an XMM register. |
| MOVD <i>xmm, reg/mem64</i> | 66 0F 6E/r | Move 64-bit value from a general-purpose register or 64-bit memory location to an XMM register. |
| MOVD <i>reg/mem32, xmm</i> | 66 0F 7E/r | Move 32-bit value from an XMM register to a general-purpose register or 32-bit memory location. |
| MOVD <i>reg/mem64, xmm</i> | 66 0F 7E/r | Move 64-bit value from an XMM register to a general-purpose register or 64-bit memory location. |
| MOVD <i>mmx, reg/mem32</i> | 0F 6E/r | Move 32-bit value from a general-purpose register or 32-bit memory location to an MMX register. |
| MOVD <i>mmx, reg/mem64</i> | 0F 6E/r | Move 64-bit value from a general-purpose register or 64-bit memory location to an MMX register. |
| MOVD <i>reg/mem32, mmx</i> | 0F 7E/r | Move 32-bit value from an MMX register to a general-purpose register or 32-bit memory location. |
| MOVD <i>reg/mem64, mmx</i> | 0F 7E/r | Move 64-bit value from an MMX register to a general-purpose register or 64-bit memory location. |

The following diagrams illustrate the operation of the MOVD instruction.

Related Instructions

MOVDQA, MOVDQU, MOVDQ2Q, MOVQ, MOVQ2DQ

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions (All Modes)

| Exception | Real | Virtual 8086 | Protected | Description |
|---|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The emulate bit (EM) of CR0 was set to 1. The MMX instructions are not supported, as indicated by bit 23 of CPUID function 1 or extended function 8000_0001. The SSE2 instructions are not supported, as indicated by bit 26 of CPUID extended function 8000_0001. |
| Device not available, #NM | X | X | X | The task-switch bit (TS) of CR0 was set to 1. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| x87 floating-point exception pending, #MF | X | X | X | An exception was pending due to an x87 floating-point instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MOVMSKPD**Extract Packed Double-Precision Floating-Point Sign Mask**

Moves the sign bits of two packed double-precision floating-point values in an XMM register (second operand) to the two low-order bits of a general-purpose register (first operand) with zero-extension.

| Mnemonic | Opcode | Description |
|----------------------------|---------------|---|
| MOVMSKPD <i>reg32, xmm</i> | 66 0F 50 /r | Move sign bits in an XMM register to a 32-bit general-purpose register. |
| MOVMSKPD <i>reg64, xmm</i> | 66 0F 50 /r | Move sign bits in an XMM register to a 64-bit general-purpose register. |

Related Instructions

MOVMSKPS, PMOVMSKB

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

| Exception (vector) | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------------|-------------|-------------------------|------------------|---|
| Invalid opcode, #UD | X | X | X | The emulate bit (EM) of CR0 was set to 1. The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0. The SSE2 instructions are not supported, as indicated by bit 26 of CPUID extended function 8000_0001. |
| Device not available, #NM | X | X | X | The task-switch bit (TS) of CR0 was set to 1. |

MOVMSKPS **Extract Packed Single-Precision Floating-Point Sign Mask**

Moves the sign bits of four packed single-precision floating-point values in an XMM register (second operand) to the four low-order bits of a general-purpose register (first operand) with zero-extension.

| Mnemonic | Opcode | Description |
|----------------------------|---------------|---|
| MOVMSKPS <i>reg32, xmm</i> | 0F 50 /r | Move sign bits in an XMM register to a 32-bit general-purpose register. |
| MOVMSKPS <i>reg64, xmm</i> | 0F 50 /r | Move sign bits in an XMM register to a 64-bit general-purpose register. |

Related Instructions

MOVMSKPD, PMOVMSKB

rFLAGS Affected

None

MXCSR Flags Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------------|-------------|-------------------------|------------------|---|
| Invalid opcode, #UD | X | X | X | The emulate bit (EM) of CR0 was set to 1. The operating-system FXSAVE/FXRSTOR support bit (OSFXSR) of CR4 was cleared to 0. The SSE2 instructions are not supported, as indicated by bit 26 of CPUID extended function 8000_0001. |
| Device not available, #NM | X | X | X | The task-switch bit (TS) of CR0 was set to 1. |

MOVNTI Move Non-Temporal Doubleword or Quadword

Stores a the value in a 32-bit or 64-bit general-purpose register (second operand) in a memory location (first operand). This instruction indicates to the processor that the data is non-temporal and is unlikely to be used again soon. The processor treats the store as a write-combining memory write, which minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see “Memory Optimization” in Volume 1.

The MOVNTI instruction is weakly-ordered with respect to other instructions that operate on memory. Software should use an SFENCE instruction to force strong memory ordering of MOVNTI with respect to other stores.

Support for the MOVNTI instruction is indicated when the SSE2 bit (bit 26) is set to 1 in EDX after executing CPUID with extended function 8000_0001h.

| Mnemonic | Opcode | Description |
|----------------------------|----------|---|
| MOVNTI <i>mem32, reg32</i> | 0F C3 /r | Stores a 32-bit general-purpose register value into a 32-bit memory location, minimizing cache pollution. |
| MOVNTI <i>mem64, reg64</i> | 0F C3 /r | Stores a 64-bit general-purpose register value into a 64-bit memory location, minimizing cache pollution. |

Related Instructions

MOVNTDQ, MOVNTPD, MOVNTPS, MOVNTQ

Exceptions

| Exception (vector) | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|--------------|-----------|---|
| Invalid opcode, #UD | X | X | X | The SSE2 instructions are not supported, as indicated by bit 26 of CPUID extended function 8000_0001. |
| Device not available, #NM | X | X | X | The task-switch bit (TS) of CR0 was set to 1. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |

MOVS **Move String**

MOVSB

MOVSW

MOVSD

MOVSQ

Moves a byte, word, doubleword, or quadword from the memory location pointed to by DS:rSI to the memory location pointed to by ES:rDI, and then increments or decrements the rSI and rDI registers according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments both pointers; otherwise, it decrements them. It increments or decrements the pointers by 1, 2, 4, or 8, depending on the size of the operands.

The forms of the MOV_Sx instruction with explicit operands address the first operand at *seg*:[rSI]. The value of *seg* defaults to the DS segment, but can be overridden by a segment prefix. These instructions always address the second operand at ES:[rDI] (ES may not be overridden). The explicit operands serve only to specify the type (size) of the value being moved.

The no-operands forms of the instruction use the DS:[rSI] and ES:[rDI] registers to point to the value to be moved (they do not allow a segment prefix). The mnemonic determines the size of the operands.

Do not confuse this MOVSD instruction with the same-mnemonic MOVSD (move scalar double-precision floating-point) instruction in the 128-bit media instruction set. Assemblers can distinguish the instructions by the number and type of operands.

The MOV_Sx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 10. The MOV_Sx instructions can also operate inside a LOOP_{cc} instruction.

| Mnemonic | Opcode | Description |
|--------------------------|---------------|--------------------------------------|
| MOVS <i>mem8, mem8</i> | A4 | Move byte at DS:SI to ES:DI. |
| MOVS <i>mem16, mem16</i> | A5 | Move word at DS:SI to ES:DI. |
| MOVS <i>mem32, mem32</i> | A5 | Move doubleword at DS:ESI to ES:EDI. |
| MOVS <i>mem64, mem64</i> | A5 | Move quadword at DS:RSI to ES:RDI. |

| Mnemonic | Opcode | Description |
|----------|--------|--------------------------------------|
| MOVSB | A4 | Move byte at DS:SI to ES:DI. |
| MOVSW | A5 | Move word at DS:SI to ES:DI. |
| MOVSD | A5 | Move doubleword at DS:ESI to ES:EDI. |
| MOVSQ | A5 | Move quadword at DS:RSI to ES:RDI. |

Related Instructions

MOVSB, MOVSD, MOVSQ, MOVSW

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MOVSX Move with Sign-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), extending the most significant bit of an 8-bit or 16-bit value into all higher bits in a 16-bit, 32-bit, or 64-bit register.

| Mnemonic | Opcode | Description |
|-------------------------------|----------|--|
| <i>MOVSX reg16, reg/mem8</i> | 0F BE /r | Move the contents of an 8-bit register or memory location to a 16-bit register with sign extension. |
| <i>MOVSX reg32, reg/mem8</i> | 0F BE /r | Move the contents of an 8-bit register or memory location to a 32-bit register with sign extension. |
| <i>MOVSX reg64, reg/mem8</i> | 0F BE /r | Move the contents of an 8-bit register or memory location to a 64-bit register with sign extension. |
| <i>MOVSX reg32, reg/mem16</i> | 0F BF /r | Move the contents of an 16-bit register or memory location to a 32-bit register with sign extension. |
| <i>MOVSX reg64, reg/mem16</i> | 0F BF /r | Move the contents of an 16-bit register or memory location to a 64-bit register with sign extension. |

Related Instructions

MOVSXD, MOV, MOVS, MOVSB, MOVSD, MOVSQ, MOVSW, MOVZX

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MOVSXD**Move with Sign-Extend Doubleword**

Copies the 32-bit value in a register or memory location (second operand) into a 64-bit register (first operand), extending the most significant bit of the 32-bit value into all higher bits of the 64-bit register.

This instruction requires a REX prefix to do the work of sign-extending a 32-bit source operand to a 64-bit result. Without a REX prefix, the operand size is 32 bits and the source is zero-extended into a 64-bit register. With a 16-bit operand size, the instruction is a NOP.

This instruction is available only in 64-bit mode. In legacy or compatibility mode this opcode is interpreted as ARPL.

| Mnemonic | Opcode | Description |
|--------------------------------|--------|--|
| MOVSXD <i>reg64, reg/mem32</i> | 63 /r | Move the contents of a 32-bit register or memory operand to a 64-bit register with sign extension. |

Related Instructions

MOVSX, MOV, MOVS, MOVSB, MOVSD, MOVSQ, MOVSW, MOVZX

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MOVZX Move with Zero-Extension

Copies the value in a register or memory location (second operand) into a register (first operand), zero-extending the value to fit in the destination register. The operand-size attribute determines the size of the zero-extended value.

This instruction assists in unsigned arithmetic operations.

| Mnemonic | Opcode | Description |
|-------------------------------|----------|---|
| MOVZX <i>reg16, reg/mem8</i> | 0F B6 /r | Move the contents of an 8-bit register or memory operand to a 16-bit register with zero-extension. |
| MOVZX <i>reg32, reg/mem8</i> | 0F B6 /r | Move the contents of an 8-bit register or memory operand to a 32-bit register with zero-extension. |
| MOVZX <i>reg64, reg/mem8</i> | 0F B6 /r | Move the contents of an 8-bit register or memory operand to a 64-bit register with zero-extension. |
| MOVZX <i>reg32, reg/mem16</i> | 0F B7 /r | Move the contents of an 16-bit register or memory operand to a 32-bit register with zero-extension. |
| MOVZX <i>reg64, reg/mem16</i> | 0F B7 /r | Move the contents of an 16-bit register or memory operand to a 64-bit register with zero-extension. |

Related Instructions

MOV, MOVS, MOVSB, MOVSD, MOVSD, MOVSW, MOVZX

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------|------|-----------------|-----------|--|
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

MUL Multiply Unsigned

Multiplies the unsigned byte, word, doubleword, or quadword value in the specified register or memory location by the value in AL, AX, EAX, or RAX and stores the result in AX, DX:AX, EDX:EAX, or RDX:RAX (depending on the operand size). It puts the high-order bits of the product in AH, DX, EDX, or RDX.

If the upper half of the product is non-zero, the instruction sets the carry flag (CF) and overflow flag (OF) both to 1. Otherwise, it clears CF and OF to 0. The other arithmetic flags (SF, ZF, AF, PF) are undefined.

| Mnemonic | Opcode | Description |
|----------------------|--------|---|
| MUL <i>reg/mem8</i> | F6 /4 | Multiplies a 8-bit register or memory operand by the contents of the AL register and stores the result in the AX register. |
| MUL <i>reg/mem16</i> | F7 /4 | Multiplies a 16-bit register or memory operand by the contents of the AX register and stores the result in the DX:AX register. |
| MUL <i>reg/mem32</i> | F7 /4 | Multiplies a 32-bit register or memory operand by the contents of the EAX register and stores the result in the EDX:EAX register. |
| MUL <i>reg/mem64</i> | F7 /4 | Multiplies a 64-bit register or memory operand by the contents of the RAX register and stores the result in the RDX:RAX register. |

Related Instructions

DIV, FIDIV, FIMUL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | U | U | U | U | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference is performed while alignment checking was enabled. |

NEG**Two's Complement Negation**

Performs the two's complement negation of the value in the specified register or memory location by subtracting the value from 0. Use this instruction only on signed integer numbers.

If the value is 0, the instruction clears the CF flag to 0; otherwise, it sets CF to 1. The OF, SF, ZF, AF, and PF flag settings depend on the result of the operation.

The memory-operand forms of the NEG instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|----------------------|--------|--|
| NEG <i>reg/mem8</i> | F6 /3 | Performs a two's complement negation on an 8-bit register or memory operand. |
| NEG <i>reg/mem16</i> | F7 /3 | Performs a two's complement negation on a 16-bit register or memory operand. |
| NEG <i>reg/mem32</i> | F7 /3 | Performs a two's complement negation on a 32-bit register or memory operand. |
| NEG <i>reg/mem64</i> | F7 /3 | Performs a two's complement negation on a 64-bit register or memory operand. |

Related Instructions

AND, NOT, OR, XOR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand is in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

NOP **No Operation**

Does nothing. This one-byte instruction increments the EIP to point to next instruction in the instruction stream, but does not affect the machine state in any other way.

The NOP instruction is an alias for `XCHG eAX,eAX`.

| Mnemonic | Opcode | Description |
|-----------------|---------------|------------------------|
| NOP | 90 | Performs no operation. |

Related Instructions

FNOP

rFLAGS Affected

None

Exceptions

None

NOT One's Complement Negation

Performs the one's complement negation of the value in the specified register or memory location by inverting each bit of the value.

The memory-operand forms of the NOT instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|----------------------|--------|--|
| NOT <i>reg/mem8</i> | F6 /2 | Complements the bits in an 8-bit register or memory operand. |
| NOT <i>reg/mem16</i> | F7 /2 | Complements the bits in a 16-bit register or memory operand. |
| NOT <i>reg/mem32</i> | F7 /2 | Complements the bits in a 32-bit register or memory operand. |
| NOT <i>reg/mem64</i> | F7 /2 | Complements the bits in a 64-bit register or memory operand. |

Related Instructions

AND, NEG, OR, XOR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference is performed while alignment checking was enabled. |

OR Logical OR

Performs a logical OR on the bits in a register, memory location, or immediate value (second operand) and a register or memory location (first operand) and stores the result in the first operand location. The two operands cannot both be memory locations. Use this instruction for turning on a bit or for testing a register value for zero.

If both corresponding bits are 0, the corresponding bit of the result is 0; otherwise, the corresponding result bit is 1.

The memory-operand forms of the OR instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| OR AL, <i>imm8</i> | 0C <i>ib</i> | OR the contents of AL with an immediate 8-bit value. |
| OR AX, <i>imm16</i> | 0D <i>iw</i> | OR the contents of AX with an immediate 16-bit value. |
| OR EAX, <i>imm32</i> | 0D <i>id</i> | OR the contents of EAX with an immediate 32-bit value. |
| OR RAX, <i>imm32</i> | 0D <i>id</i> | OR the contents of RAX with a sign-extended immediate 32-bit value. |
| OR <i>reg/mem8</i> , <i>imm8</i> | 80 /1 <i>ib</i> | OR the contents of an 8-bit register or memory operand and an immediate 8-bit value. |
| OR <i>reg/mem16</i> , <i>imm16</i> | 81 /1 <i>iw</i> | OR the contents of a 16-bit register or memory operand and an immediate 16-bit value. |
| OR <i>reg/mem32</i> , <i>imm32</i> | 81 /1 <i>id</i> | OR the contents of a 32-bit register or memory operand and an immediate 32-bit value. |
| OR <i>reg/mem64</i> , <i>imm32</i> | 81 /1 <i>id</i> | OR the contents of a 64-bit register or memory operand and sign-extended immediate 32-bit value. |
| OR <i>reg/mem16</i> , <i>imm8</i> | 83 /1 <i>ib</i> | OR the contents of a 16-bit register or memory operand and a sign-extended immediate 8-bit value. |
| OR <i>reg/mem32</i> , <i>imm8</i> | 83 /1 <i>ib</i> | OR the contents of a 32-bit register or memory operand and a sign-extended immediate 8-bit value. |
| OR <i>reg/mem64</i> , <i>imm8</i> | 83 /1 <i>ib</i> | OR the contents of a 64-bit register or memory operand and a sign-extended immediate 8-bit value. |
| OR <i>reg/mem8</i> , <i>reg8</i> | 08 / <i>r</i> | OR the contents of an 8-bit register or memory operand with the contents of an 8-bit register. |

| Mnemonic | Opcode | Description |
|---------------------|--------|--|
| OR reg/mem16, reg16 | 09/r | OR the contents of a 16-bit register or memory operand with the contents of a 16-bit register. |
| OR reg/mem32, reg32 | 09/r | OR the contents of a 32-bit register or memory operand with the contents of a 32-bit register. |
| OR reg/mem64, reg64 | 09/r | OR the contents of a 64-bit register or memory operand with the contents of a 64-bit register. |
| OR reg8, reg/mem8 | 0A/r | OR the contents of an 8-bit register with the contents of an 8-bit register or memory operand. |
| OR reg16, reg/mem16 | 0B/r | OR the contents of a 16-bit register with the contents of a 16-bit register or memory operand. |
| OR reg32, reg/mem32 | 0B/r | OR the contents of a 32-bit register with the contents of a 32-bit register or memory operand. |
| OR reg64, reg/mem64 | 0B/r | OR the contents of a 64-bit register with the contents of a 64-bit register or memory operand. |

The following chart summarizes the effect of this instruction:

| X | Y | X OR Y |
|---|---|--------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Related Instructions

AND, NEG, NOT, XOR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|----|-----|----|
| | | | | | | | | 0 | | | | 1/0 | 1/0 | U | 1/0 | 0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

OUT Output to Port

Copies the value from the AL, AX, or EAX register (second operand) to an I/O port (first operand). The port address can be a byte-immediate value (00h to FFh) or the value in the DX register (0000h to FFFFh). The register used determines the size of the port (8, 16, or 32 bits).

| Mnemonic | Opcode | Description |
|-----------------------|--------------|--|
| OUT <i>imm8</i> , AL | E6 <i>ib</i> | Output the byte in the AL register to the port specified by an 8-bit immediate value. |
| OUT <i>imm8</i> , AX | E7 <i>ib</i> | Output the word in the AX register to the port specified by an 8-bit immediate value. |
| OUT <i>imm8</i> , EAX | E7 <i>ib</i> | Output the doubleword in the EAX register to the port specified by an 8-bit immediate value. |
| OUT DX, AL | EE | Output byte in AL to the output port specified in DX. |
| OUT DX, AX | EF | Output byte in AX to the output port specified in DX. |
| OUT DX, EAX | EF | Output byte in EAX to the output port specified in DX. |

Related Instructions

IN

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|---|
| General protection, #GP | | X | X | At least one of the corresponding I/O permission bits was 1 in the TSS for the accessed I/O port. The current privilege level was greater than the I/O privilege level of the current task and at least one of the corresponding I/O permission bits was 1 in the TSS for the accessed I/O port. |

OUTS Output String

OUTSB

OUTSD

OUTSW

Copies data from the memory location pointed to by DS:eSI to the I/O port address (0000h to FFFFh) specified in the DX register and increments or decrements the eSI register according to the setting of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments eSI; otherwise, it decrements eSI. It increments or decrements the pointer by 1, 2, or 4, depending on the size of the value being copied.

The OUTS instruction uses an explicit memory operand (second operand) to determine the type (size) of the value being copied, but always uses DS:eSI for the location of the value to copy. The explicit register operand specifies the I/O port address and must always be DX.

The no-operands forms of the instruction use the DS:[eSI] register pair to point to the data to be copied and the DX register as the destination. The mnemonic specifies the size of the I/O port and the type (size) of the value being copied.

The OUTSx instructions support the REP prefix. For details about the REP prefix, see “Repeat Prefixes” on page 10.

| Mnemonic | Opcode | Description |
|-----------------------|--------|--|
| OUTS DX, <i>mem8</i> | 6E | Output the byte in DS:SI to the port specified in DX. |
| OUTS DX, <i>mem16</i> | 6F | Output the word in DS:SI to the port specified in DX. |
| OUTS DX, <i>mem32</i> | 6F | Output the doubleword in DS:ESI to the port specified in DX. |
| OUTSB | 6E | Output the byte in DS:SI to the port specified in DX. |
| OUTSW | 6F | Output the word in DS:SI to the port specified in DX. |
| OUTSD | 6F | Output the doubleword in DS:ESI to the port specified in DX. |

Related Instructions

INS, OUTSB, OUTSD, OUTSW

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | | | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | At least one of the corresponding I/O permission bits was 1 in the TSS for the accessed I/O port. |
| | | | X | The segment register contained a null segment selector. |
| | | | X | The current privilege level was greater than the I/O privilege level of the current task and at least one of the corresponding I/O permission bits was 1 in the TSS for the accessed I/O port. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference is performed while alignment checking was enabled. |

POP**Pop Stack**

Copies the value pointed to by the stack pointer to the specified register or memory location and increments the SP by 2 for the 16-bit stack, 4 for a 32-bit stack, or 8 for a 64-bit stack.

The address-size attribute of the stack segment determines the size of the stack pointer. The operand-size attribute of the current code segment determines the amount by which the stack pointer is incremented (2,4 or 8 bytes).

For forms of the instruction that load a segment register (POP DS, POP ES, POP FS, POP GS, POP SS), the source operand must be a valid segment selector.

It is possible to pop a null segment selector value (0000_0003h) into the DS, ES, FS, or GS register (but not into SS!). This action does not cause a general protection fault, but a subsequent reference to such a segment *does* cause a #GP exception. For more information about segment selectors, see “Segment Selectors and Registers” on page 82.

In protected mode, when a segment selector is popped into a segment register, the processor also loads all associated descriptor information into the hidden part of the register and validates it.

In 64-bit mode, the operand size defaults to 64 bits and there is no prefix available for encoding a 32-bit operand size. Using the 1F, 07, or 17 opcode of this instruction in 64-bit mode generates an invalid-opcode exception.

This instruction cannot pop a value into the CS register. The RET instruction performs this function.

| Mnemonic | Opcode | Description |
|-----------------|---------------|--|
| POP reg/mem16 | 8F /0 | Pop the top of the stack into a register or 16-bit memory location and increment the stack pointer. |
| POP reg/mem32 | 8F /0 | Pop the top of the stack into a register or 32-bit memory location and increment the stack pointer. (No prefix for encoding this in 64-bit mode.) |
| POP reg/mem64 | 8F /0 | Pop the top of the stack into a register or 64-bit memory location and increment the stack pointer. |
| POP reg16 | 58 +rw | Pop the top of the stack into a 16-bit register and increment the stack pointer. |

| Mnemonic | Opcode | Description |
|------------------|---------------|---|
| POP <i>reg32</i> | 58 <i>rd</i> | Pop the top of the stack into a 32-bit register and increment the stack pointer. (No prefix for encoding this in 64-bit mode.) |
| POP <i>reg64</i> | 58 <i>rq</i> | Pop the top of the stack into a 64-bit register and increment the stack pointer. |
| POP DS | 1F | Pop the top of the stack into the DS register and increment the stack pointer. (Invalid in 64-bit mode.) |
| POP ES | 07 | Pop the top of the stack into the ES register and increment the stack pointer. (Invalid in 64-bit mode.) |
| POP SS | 17 | Pop the top of the stack into the SS register and increment the stack pointer. (Invalid in 64-bit mode.) |
| POP FS | 0F A1 | Pop the top of the stack into the FS register and increment the stack pointer. |
| POP GS | 0F A9 | Pop the top of the stack into the GS register and increment the stack pointer. |

Related Instructions

FPOP, FPUSH, PUSH

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|-------------|-------------------------|------------------|--|
| Invalid opcode, #UD | | | X | The 1F, 07, and 17 opcodes of this instruction are not recognized in 64-bit mode. |
| Segment not present, #NP (selector) | | | X | The DS, ES, FS, or GS register was loaded with a non-null segment selector and the segment was marked not present. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | | | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | An attempt was made to load SS with the null segment selector. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| General protection, #GP (selector) | | | X | The segment selector index was outside the descriptor table limits. |
| | | | X | The SS register was being loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL. |
| | | | X | The SS register was being loaded and the segment pointed to was a non-writable data segment. |
| | | | X | The DS, ES, FS, or GS register was being loaded and the segment pointed to was not a data or readable code segment. |
| | | | X | The DS, ES, FS, or GS register was being loaded and the segment pointed to was a data or nonconforming code segment, but both the RPL and the CPL were greater than the DPL. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

POPA POPAD

POP All to GPR

Pops words or doublewords from the stack into the general-purpose registers in the following order: (E)DI, (E)SI, (E)BP, (E)BX, (E)DX, (E)CX, and (E)AX (the (E)SP register value on the stack is discarded). The instruction increments the stack pointer by 2 or 4 after it loads each register, depending on the mnemonic.

Using the POPA or POPAD instructions in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| POPA | 61 | Pop the DI, SI, BP, SP, BX, DX, CX, and AX registers. (Invalid in 64-bit mode.) |
| POPAD | 61 | Pop the EDI, ESI, EBP, ESP, EBX, EDX, ECX, and EAX registers. (Invalid in 64-bit mode.) |

Related Instructions

FPOP, FPUSH, PUSH

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | The 1F, 07, and 17 opcodes of this instruction are not recognized in 64-bit mode. |
| Segment not present, #NP (selector) | | | X | The DS, ES, FS, or GS register was being loaded with a non-null segment selector and the segment was marked not present. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | | | As part of a stack switch, the SS register was loaded but the specified segment was marked as not present. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|------------------------------------|------|-----------------|------------------|---|
| General protection, #GP | | | X X X X | A memory address exceeded a data segment limit or was non-canonical. An attempt was made to load SS with the null segment selector. The destination operand was in a non-writable segment. A null data segment was being used to reference memory. |
| General protection, #GP (selector) | | | X X X X | The segment selector index was outside the descriptor table limits. The SS register was being loaded and the segment selector RPL and the segment descriptor DPL were not equal to the CPL. The SS register was being loaded and the segment pointed to was a non-writable data segment. The DS, ES, FS, or GS register was being loaded and the segment pointed to was not a data or readable code segment. The DS, ES, FS, or GS register was being loaded and the segment pointed to was a data or nonconforming code segment, but both the RPL and the CPL were greater than the DPL. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

POPF

POPFD

POPFBQ

POP to rFLAGS

Pops a word, doubleword, or quadword from the stack into the rFLAGS register and increments the stack pointer by 2, 4, or 8, depending on the mnemonic.

In protected or real-address mode at privilege level 0, all the non-reserved flags in the rFLAGS register can be modified, except the VIP and VIF flags, which are cleared, and the VM flag, which remains unaffected. At a privilege level greater than 0, but less than or equal to IOPL, the IOPL and VM flags are unaffected and the VIP and VIF flags are cleared.

In virtual-8086 mode, if IOPL field is less than 3, attempting to execute a POPF, POPFD, POPFBQ, PUSHF, PUSHFD, or PUSHFBQ instruction generates a #GP exception.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| POPF | 9D | Pop a word from the stack into the FLAGS register (the lower 16-bits of the EFLAGS register) and increment SP by 2. |
| POPFD | 9D | Pop a double word from the stack into the EFLAGS register and increment ESP by 4. |
| POPFBQ | 9D | Pop a quadword from the stack to the RFLAGS register and increment RSP by 8. |

Action

// See “Pseudocode Definitions” on page 46.

POPF_START:

```
IF (REAL_MODE)
    POPF_REAL
ELIF (PROTECTED_MODE)
    POPF_PROTECTED
ELSE // (VIRTUAL_MODE)
    POPF_VIRTUAL
```

POPF_REAL:

```
POP.v temp_RFLAGS
RFLAGS.v = temp_RFLAGS // VIF,VIP,VM unchanged
```

```

// RF cleared
EXIT

POPF_PROTECTED:

POP.v temp_RFLAGS
RFLAGS.v = temp_RFLAGS          // VIF,VIP,VM unchanged
                                // IOPL changed only if (CPL=0)
                                // IF changed only if (CPL<=old_RFLAGS.IOPL)
                                // RF cleared

EXIT

POPF_VIRTUAL:

IF (RFLAGS.IOPL=3)
{
    POP.v temp_RFLAGS
    RFLAGS.v = temp_RFLAGS      // VIF,VIP,VM,IOPL unchanged
                                // RF cleared

    EXIT
}
ELIF ((CR4.VME=1) && (OPERAND_SIZE=16))
{
    POP.w temp_RFLAGS
    IF (((temp_RFLAGS.IF=1) && (RFLAGS.VIP=1)) || (temp_RFLAGS.TF=1))
        EXCEPTION [#GP(0)]
                                // notify the virtual-mode-manager to deliver
                                // the task's pending interrupts

    RFLAGS.w = temp_RFLAGS      // IF,IOPL unchanged
                                // RFLAGS.VIF=temp_RFLAGS.IF
                                // RF cleared

    EXIT
}
ELSE // (((RFLAGS.IOPL<3) && ((CR4.VME=0) || (OPERAND_SIZE!=16)))
    EXCEPTION [#GP(0)]

```

Related Instructions

POP, POPA, POPAD, PUSHF, PUSHFD, PUSHFQ

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|-----|----|----|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1/0 | | | 1/0 | | 0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | X X | | The I/O privilege level was less than 3. An attempt was made to execute this instruction with an operand-size override prefix. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

PREFETCH PREFETCHW

Prefetch L1 Data-Cache Line

PREFETCH and PREFETCHW are 3DNow!™ instructions. They load a cache line into the L1 data cache from the specified memory address. The PREFETCH instruction loads a cache line even if the *mem8* address is not aligned with the start of the line. If a cache hit occurs, or if a memory fault is detected, no bus cycle is initiated, and the instruction is treated as a NOP.

The PREFETCHW instruction loads the prefetched line and sets the cache-line state to Modified, in anticipation of subsequent data writes to the line. The PREFETCH instruction, by contrast, typically (depending on hardware implementation) sets the cache-line state to Exclusive.

The opcodes for the instructions include the ModRM byte, and only the memory form of ModRM is valid. The register form of ModRM causes an invalid-opcode exception. Because there is no destination register, the three destination register field bits of the ModRM byte define the type of prefetch to be performed. The bit patterns 000b and 001b define the PREFETCH and PREFETCHW instructions, respectively. All other bit patterns are reserved for future use.

Table 3-15 on page 221 summarizes the PREFETCH type options. The *reserved* PREFETCH types do not result in an invalid-opcode exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all reserved PREFETCH types are implemented as synonyms of the basic PREFETCH type (the PREFETCH instruction with type 000b).

The operation of these instructions is implementation-dependent. The processor implementation can ignore or change these instructions. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. For details on the use of this instruction, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

| Mnemonic | Opcode | Description |
|-----------------------|----------|---|
| PREFETCH <i>mem8</i> | OF 0D /0 | Prefetch processor cache line into L1 data cache. |
| PREFETCHW <i>mem8</i> | OF 0D /1 | Prefetch processor cache line into L1 data cache. |

Table 3-15. Summary of PREFETCH Instruction Type Options

| ModRM Byte | Result ¹ |
|--|---------------------|
| 11-xxx-xxx | Invalid opcode |
| mm-000-xxx | PREFETCH |
| mm-001-xxx | PREFETCHW |
| mm-010-xxx | <i>reserved</i> |
| mm-011-xxx | <i>reserved</i> |
| mm-100-xxx | <i>reserved</i> |
| mm-101-xxx | <i>reserved</i> |
| mm-110-xxx | <i>reserved</i> |
| mm-111-xxx | <i>reserved</i> |
| Note: 1. The "Reserved" PREFETCH types do not result in an Invalid Opcode Exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all "Reserved" PREFETCH types are implemented as synonyms for the basic PREFETCH type (the PREFETCH instruction with type 000b). | |

Related InstructionsPREFETCH_{level}**rFLAGS Affected**

None

Exceptions

| Exception (vector) | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|--------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The AMD 3DNow!™ instructions are not supported, as indicated by bit 31 of CPUID extended function 8000_0001. |

PREFETCH_{level}**Prefetch Data to Cache Level _{level}**

Loads a cache line from the specified memory address into the data-cache level specified by the locality reference bits 5–3 of the ModRM byte. Table 3-16 on page 223 lists the locality reference options for the instruction.

This instruction loads a cache line even if the *mem8* address is not aligned with the start of the line. If the cache line is already contained in a cache level that is lower than the specified locality reference, or if a memory fault is detected, a bus cycle is not initiated and the instruction is treated as a NOP.

The operation of this instruction is implementation-dependent. The processor implementation can ignore or change this instruction. The size of the cache line also depends on the implementation, with a minimum size of 32 bytes. For details on the use of this instruction, see the data sheet or other software-optimization documentation relating to particular hardware implementations.

| Mnemonic | Opcode | Description |
|-------------------------|---------------|--|
| PREFETCHNTA <i>mem8</i> | 0F 18 /0 | Move data closer to the processor using the NTA reference. |
| PREFETCHT0 <i>mem8</i> | 0F 18 /1 | Move data closer to the processor using the T0 reference. |
| PREFETCHT1 <i>mem8</i> | 0F 18 /2 | Move data closer to the processor using the T1 reference. |
| PREFETCHT2 <i>mem8</i> | 0F 18 /3 | Move data closer to the processor using the T2 reference. |

Table 3-16. Locality References for the Prefetch Instructions

| Locality Reference | Description |
|--------------------|---|
| NTA | Non-Temporal Access—Move the specified data into the processor with minimum cache pollution. This is intended for data that will be used only once, rather than repeatedly. The specific technique for minimizing cache pollution is implementation-dependent and may include such techniques as allocating space in a software-invisible buffer, allocating a cache line in only a single way, etc. For details, see the software-optimization documentation for a particular hardware implementation. |
| T0 | All Cache Levels—Move the specified data into all cache levels. |
| T1 | Level 2 and Higher—Move the specified data into all cache levels except 0th level (L1) cache. |
| T2 | Level 3 and Higher—Move the specified data into all cache levels except 0th level (L1) and 1st level (L2) caches. |

Related Instructions

PREFETCH, PREFETCHW

rFLAGS Affected

None

Exceptions

None

PUSH**Push onto Stack**

Decrements the stack pointer and then copies the specified immediate value or the value in the specified register or memory location to the top of the stack (the memory location pointed to by SS:rSP).

If the instruction pushes the stack pointer (rSP), the resulting value on the stack is that of rSP before execution of the instruction.

There is a PUSH CS instruction but no corresponding POP CS. The RET instruction pops a value from the top of stack into the CS register as part of its operation.

In 64-bit mode, the operand size of all PUSH instructions defaults to 64 bits, and there is no prefix available for encoding a 32-bit operand size. Using the 0E, 16, 1E, or 0F opcodes of this instruction in 64-bit mode generates an invalid-opcode exception.

Pushing an odd number of 16-bit operands when the stack address-size attribute is 32 results in a misaligned stack pointer.

| Mnemonic | Opcode | Description |
|-----------------------|----------------|--|
| PUSH <i>reg/mem16</i> | FF /6 | Push the contents of a 16-bit register or memory operand onto the stack and decrement the stack pointer. |
| PUSH <i>reg/mem32</i> | FF /6 | Push the contents of a 32-bit register or memory operand onto the stack and decrement the stack pointer. |
| PUSH <i>reg/mem64</i> | FF /6 | Push the contents of a 64-bit register or memory operand onto the stack and decrement the stack pointer. |
| PUSH <i>reg16</i> | 50 + <i>rw</i> | Push the contents of a 16-bit register onto the stack and decrement the stack pointer. |
| PUSH <i>reg32</i> | 50 + <i>rd</i> | Push the contents of a 32-bit register onto the stack and decrement the stack pointer. |
| PUSH <i>reg64</i> | 50 + <i>rq</i> | Push the contents of a 64-bit register onto the stack and decrement the stack pointer. |
| PUSH <i>imm8</i> | 6A | Push an 8-bit immediate value onto the stack and decrement the stack pointer. |
| PUSH <i>imm16</i> | 68 | Push a 16-bit immediate value onto the stack and decrement the stack pointer. |
| PUSH <i>imm32</i> | 68 | Push a 32-bit immediate value onto the stack and decrement the stack pointer. |

| Mnemonic | Opcode | Description |
|-----------------|---------------|--|
| PUSH CS | 0E | Push the contents of the CS register onto the stack and decrement the stack pointer. (Invalid in 64-bit mode.) |
| PUSH SS | 16 | Push the contents of the SS register onto the stack and decrement the stack pointer. (Invalid in 64-bit mode.) |
| PUSH DS | 1E | Push the contents of the DS register onto the stack and decrement the stack pointer. (Invalid in 64-bit mode.) |
| PUSH ES | 06 | Push the contents of the ES register onto the stack and decrement the stack pointer. (Invalid in 64-bit mode.) |
| PUSH FS | 0F A0 | Push the contents of the FS register onto the stack and decrement the stack pointer. |
| PUSH GS | 0F A8 | Push the contents of the GS register onto the stack and decrement the stack pointer. |

Related Instructions

PUSHA, PUSHAD, PUSHAX, PUSHF, PUSHFD, PUSHFQ, POP, POPA, POPAD, POPE, POPFD, PUSHFQ

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|---|
| Invalid opcode, #UD | | | X | The 0E, 1E, 06, and 16 opcodes of this instruction are not recognized in 64-bit mode. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| | X | | | The new value of the rSP register exceeded the stack segment limit. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | A null data segment was used to reference memory.. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------|------|-----------------|-----------|--|
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

PUSHA

PUSHAD

Push All GPRs onto Stack

Pushes the contents of the eAX, eCX, eDX, eBX, eSP (original value), eBP, eSI, and eDI general-purpose registers onto the stack in that order.

Using the PUSHA or PUSHAD instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|-----------------|---------------|---|
| PUSHA | 60 | Push the contents of the AX, CX, DX, BX, original SP, BP, SI, and DI registers onto the stack. (Invalid in 64-bit mode.) |
| PUSHAD | 60 | Push the contents of the EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI registers onto the stack. (Invalid in 64-bit mode.) |

Related Instructions

PUSH, PUSHF, PUSHFD, PUSHFQ, POP, POPA, POPAD, POPF, POPFD, PUSHFQ

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|-------------|-------------------------|------------------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, segment overrun, #GP | X | X | | The ESP or SP register contained 7, 9, 11, 13, or 15. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

PUSHF Push rFLAGS Word onto Stack

PUSHFD

PUSHFQ

Decrements the rSP register and copies the entire contents of the rFLAGS register (except for the VM and RF flags) onto the stack. The instruction clears the VM and RF flags in the rFLAGS image before putting it on the stack.

The instruction decrements the stack pointer by 2, 4, or 8, depending on the mnemonic.

When operating in virtual-8086 mode, if system software has set the IOPL field to a value less than 3, a general-protection exception occurs if application software attempts to execute a POPF, POPFD, POPFQ, PUSHF, PUSHFD, or PUSHFQ instruction. If system software has set the IOPL field to a value of 3, application software can cause the interrupt flag, rFLAGS.IF (bit 9) to be cleared to 0 by executing a CLI instruction. For a description on how this is done, and for information about how system software accesses the system bits in rFLAGS (those bits that can be accessed only by system software), see “Exceptions and Interrupts” in Volume 2.

| Mnemonic | Opcode | Description |
|-----------------|---------------|---|
| PUSHF | 9C | Pushes the FLAGS register onto the stack. |
| PUSHFD | 9C | Pushes eFLAGS doubleword onto stack. |
| PUSHFQ | 9C | Pushes rFLAGS doubleword onto stack. |

Action

// See “Pseudocode Definitions” on page 46.

PUSHF_START:

```
IF (REAL_MODE)
    PUSHF_REAL
ELIF (PROTECTED_MODE)
    PUSHF_PROTECTED
ELSE // (VIRTUAL_MODE)
    PUSHF_VIRTUAL
```

PUSHF_REAL:

```
PUSH.v old_RFLAGS    // pushed with RF clear
```



```
EXIT
```

```
PUSHF_PROTECTED:
```

```
    PUSH.v old_RFLAGS    // pushed with RF clear
    EXIT
```

```
PUSHF_VIRTUAL:
```

```
    IF (RFLAGS.IOPL=3)
    {
        PUSH.v old_RFLAGS // pushed with RF,VM clear
        EXIT
    }
    ELIF ((CR4.VME=1) && (OPERAND_SIZE=16))
    {
        PUSH.v old_RFLAGS // pushed with VIF in the IF position
                          // pushed with IOPL=3
        EXIT
    }
    ELSE // ((RFLAGS.IOPL<3) && ((CR4.VME=0) || (OPERAND_SIZE!=16)))
        EXCEPTION [#GP(0)]
```

Related Instructions

PUSHA, PUSHAD, PUSHAX, PUSHF, PUSHFD, PUSHFQ, POP, POPA, POPAD, POPE, POPFD

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | | | X | The new value of the rSP register exceeded the stack segment limit. |
| General protection, #GP | | X | | The I/O privilege level was less than 3. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

RCL Rotate Through Carry Left

Rotates the bits of a register or memory location (first operand) to the left (more significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A rotate of 1 has a separate opcode encoding.

For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is not affected.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| RCL <i>reg/mem8</i> , 1 | D0 /2 | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left once. |
| RCL <i>reg/mem8</i> , CL | D2 /2 | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified in the CL register. |
| RCL <i>reg/mem8</i> , <i>imm8</i> | C0 /2 <i>ib</i> | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value. |
| RCL <i>reg/mem16</i> , 1 | D1 /2 | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left once. |
| RCL <i>reg/mem16</i> , CL | D3 /2 | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified in the CL register. |
| RCL <i>reg/mem16</i> , <i>imm8</i> | C1 /2 <i>ib</i> | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value. |
| RCL <i>reg/mem32</i> , 1 | D1 /2 | Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left once. |
| RCL <i>reg/mem32</i> , CL | D3 /2 | Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified in the CL register. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|--|
| RCL <i>reg/mem32, imm8</i> | C1 /2 <i>ib</i> | Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value. |
| RCL <i>reg/mem64, 1</i> | D1 /2 | Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left once. |
| RCL <i>reg/mem64, CL</i> | D3 /2 | Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified in the CL register. |
| RCL <i>reg/mem64, imm8</i> | C1 /2 <i>ib</i> | Rotates the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value. |

Related Instructions

RCR, ROL, ROR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | | | | | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X X X | A memory address exceeded a data segment limit or was non-canonical. The source operand was in a non-writable segment. A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

RCR Rotate Through Carry Right

Rotates the bits of a register or memory location (first operand) to the right (less significant bit positions) and through the carry flag by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated through the carry flag are rotated back in at the left end (msb) of the first operand location.

The processor masks the upper three bits in the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A rotate of 1 has a separate opcode encoding.

For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF flag (before the rotate) and the most significant bit of the original value. When the rotate count is greater than 1, the OF flag is not affected.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|--|
| RCR <i>reg/mem8</i> , 1 | D0 /3 | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right once. |
| RCR <i>reg/mem8</i> , CL | D2 /3 | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified in the CL register. |
| RCR <i>reg/mem8</i> , <i>imm8</i> | C0 /3 <i>ib</i> | Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| RCR <i>reg/mem16</i> , 1 | D1 /3 | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right once. |
| RCR <i>reg/mem16</i> , CL | D3 /3 | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified in the CL register. |
| RCR <i>reg/mem16</i> , <i>imm8</i> | C1 /3 <i>ib</i> | Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| RCR <i>reg/mem32</i> , 1 | D1 /3 | Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right once. |
| RCR <i>reg/mem32</i> , CL | D3 /3 | Rotate 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified in the CL register. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|--|
| RCR <i>reg/mem32, imm8</i> | C1 /3 <i>ib</i> | Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| RCR <i>reg/mem64,1</i> | D1 /3 | Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right once. |
| RCR <i>reg/mem64,CL</i> | D3 /3 | Rotate 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified in the CL register. |
| RCR <i>reg/mem64, imm8</i> | C1 /3 <i>ib</i> | Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified by an 8-bit immediate value. |

Related Instructions

RCL, ROR, ROL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | | | | | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The source operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

RET**Return from Called Procedure**

Returns from a procedure previously entered by a CALL instruction. The RET instruction can execute three different types of returns:

- Near return—Returns to a calling procedure within the current code segment (pointed to by the CS register), also known as an intrasegment return.
- Far return—Returns to a calling procedure in a different segment than the current code segment, also known as an intersegment return.
- Interprivilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

For near returns, the instruction restores the rIP from the stack and increments the SP register by 2. For far return, the instruction also restores the CS value from the stack and again increments SP by 2.

The RET instruction may be followed by an immediate value that it adds to the SP after it increments the SP. This action skips over any parameters being passed back to the calling program segment that are no longer needed.

In 64-bit mode, for near returns only (not far returns), the operand size defaults to 64 bits without the need for a REX prefix.

For details about control-flow instructions, see “Control Transfers” in Volume 1, and “Control-Transfer Privilege Checks” in Volume 2.

| Mnemonic | Opcode | Description |
|------------------|---------------|--|
| RET | C3 | Near return to the calling procedure. |
| RET | CB | Far return to the calling procedure. |
| RET <i>imm16</i> | C2 <i>iw</i> | Near return to the calling procedure, with pop of the number of bytes specified by an immediate word operand from the stack. |
| RET <i>imm16</i> | CA <i>iw</i> | Far return to the calling procedure, with pop of the number of bytes specified by an immediate word operand from the stack. |

Action

```
// Far returns (RETF)
// See “Pseudocode Definitions” on page 46.
```

```
RETF_START:
```

```
IF (REAL_MODE)
```

```

    RETF_REAL_OR_VIRTUAL
ELSIF (PROTECTED_MODE)
    RETF_PROTECTED
ELSE // (VIRTUAL_MODE)
    RETF_REAL_OR_VIRTUAL

```

RETF_REAL_OR_VIRTUAL:

```

    IF (OPCODE = retf imm16)
        temp_IMM = word-sized immediate specified in the instruction,
                    zero-extended to 64 bits
    ELSE // (OPCODE = retf)
        temp_IMM = 0

    POP.v temp_RIP
    POP.v temp_CS

    IF (temp_RIP > CS.limit)
        EXCEPTION [#GP(0)]

    CS.sel = temp_CS
    CS.base = temp_CS SHL 4

    RSP.s = RSP + temp_IMM
    RIP = temp_RIP
    EXIT

```

RETF_PROTECTED:

```

    IF (OPCODE = retf imm16)
        temp_IMM = word-sized immediate specified in the instruction,
                    zero-extended to 64 bits
    ELSE // (OPCODE = retf)
        temp_IMM = 0

    POP.v temp_RIP
    POP.v temp_CS

    temp_CPL = temp_CS.rpl

    IF (CPL=temp_CPL)
    {
        CS = READ_DESCRIPTOR (temp_CS, iret_chk)

        RSP.s = RSP + temp_IMM

        IF ((64BIT_MODE) && (temp_RIP is non-canonical)
            || (!64BIT_MODE) && (temp_RIP > CS.limit))
            EXCEPTION [#GP(0)]
    }

```

```

        RIP = temp_RIP
        EXIT
    }
    ELSE // (CPL!=temp_CPL)
    {
        RSP.s = RSP + temp_IMM

        POP.v temp_RSP
        POP.v temp_SS

        CS = READ_DESCRIPTOR (temp_CS, iret_chk)

        CPL = temp_CPL

        IF ((64BIT_MODE) && (temp_RIP is non-canonical)
            || (!64BIT_MODE) && (temp_RIP > CS.limit))
            EXCEPTION [#GP(0)]

        SS = READ_DESCRIPTOR (temp_SS, ss_chk)

        IF ((DS.attr.dpl < CPL)
            && ((DS.attr.type = 'data') || (DS.attr.type = 'non-conforming-code'))))
            DS = NULL // can't use lower dpl data segment at higher cpl

        IF ((ES.attr.dpl < CPL)
            && ((ES.attr.type = 'data') || (ES.attr.type = 'non-conforming-code'))))
            ES = NULL // can't use lower dpl data segment at higher cpl

        IF ((FS.attr.dpl < CPL)
            && ((FS.attr.type = 'data') || (FS.attr.type = 'non-conforming-code'))))
            FS = NULL // can't use lower dpl data segment at higher cpl

        IF ((GS.attr.dpl < CPL)
            && ((GS.attr.type = 'data') || (GS.attr.type = 'non-conforming-code'))))
            GS = NULL // can't use lower dpl data segment at higher cpl

        RSP.s = temp_RSP
        RIP = temp_RIP
        EXIT
    }

```

Related Instructions

CALL

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| Segment not present, #NP (selector) | | | X | The return code or stack segment was not present. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| | | | X | The return stack segment was not present. |
| General protection, #GP | | | X | The return instruction pointer was not within the return code segment limit. |
| | | | X | The return code or stack segment selector was null. |
| General protection, #GP (selector) | | | X | The return code or the stack segment selector index was outside the descriptor table limits. |
| | | | X | The RPL of the return code segment selector was less than the CPL. |
| | | | X | The return code segment was nonconforming and the segment selector's DPL was not equal to the RPL of the code segment's segment selector. |
| | | | X | The return code segment was conforming and the segment selector's DPL was greater than the RPL of the code segment's segment selector. |
| | | | X | The stack segment descriptor DPL was not equal to the RPL of the return code segment selector. |
| | | | X | The stack segment was not a writable data segment. |
| | | | X | The stack segment selector RPL was not equal to the RPL of the return code segment selector. |
| General protection, segment overrun, #GP | X | X | | The segment descriptor for the return code did not indicate that it is a code segment. |
| | | | | The return address pointer fell outside the address range of 00000h to FFFFFh. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

ROL Rotate Left

Rotates the bits of a register or memory location (first operand) to the left (more significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out left are rotated back in at the right end (lsb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A rotate of 1 has a separate opcode encoding.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the lsb of the result). For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is not affected.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| ROL <i>reg/mem8</i> , 1 | D0 /0 | Rotate an 8-bit register or memory operand left once. |
| ROL <i>reg/mem8</i> , CL | D2 /0 | Rotate an 8-bit register or memory operand left the number of bits specified in the CL register. |
| ROL <i>reg/mem8</i> , <i>imm8</i> | C0 /0 <i>ib</i> | Rotate an 8-bit register or memory operand left the number of bits specified by an 8-bit immediate value. |
| ROL <i>reg/mem16</i> , 1 | D1 /0 | Rotate a 16-bit register or memory operand left once. |
| ROL <i>reg/mem16</i> , CL | D3 /0 | Rotate a 16-bit register or memory operand left the number of bits specified in the CL register. |
| ROL <i>reg/mem16</i> , <i>imm8</i> | C1 /0 <i>ib</i> | Rotate a 16-bit register or memory operand left the number of bits specified by an 8-bit immediate value. |
| ROL <i>reg/mem32</i> , 1 | D1 /0 | Rotate a 32-bit register or memory operand left once. |
| ROL <i>reg/mem32</i> , CL | D3 /0 | Rotate a 16-bit register or memory operand left the number of bits specified in the CL register. |
| ROL <i>reg/mem32</i> , <i>imm8</i> | C1 /0 <i>ib</i> | Rotate a 16-bit register or memory operand left the number of bits specified by an 8-bit immediate value. |
| ROL <i>reg/mem64</i> , 1 | D1 /0 | Rotate a 64-bit register or memory operand left once. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|---|
| ROL <i>reg/mem64, CL</i> | D3 /0 | Rotate a 64-bit register or memory operand left the number of bits specified in the CL register. |
| ROL <i>reg/mem64, imm8</i> | C1 /0 <i>ib</i> | Rotate a 64-bit register or memory operand left the number of bits specified by an 8-bit immediate value. |

Related Instructions

RCL, RCR, ROR, SHR, SHL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | | | | | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
 Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The source operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

ROR Rotate Right

Rotates the bits of a register or memory location (first operand) to the right (less significant bit positions) by the number of bit positions in an unsigned immediate value or the CL register (second operand). The bits rotated out right are rotated back in at the left end (msb) of the first operand location.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A rotate of 1 has a separate opcode encoding.

After completing the rotation, the instruction sets the CF flag to the last bit rotated out (the msb of the result). For 1-bit rotates, the instruction sets the OF flag to the exclusive OR of the two most significant bits of the result. When the rotate count is greater than 1, the OF flag is not affected.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| ROR <i>reg/mem8</i> , 1 | D0 /1 | Rotate an 8-bit register or memory location right once. |
| ROR <i>reg/mem8</i> , CL | D2 /1 | Rotate an 8-bit register or memory location right the number of bits specified in the CL register. |
| ROR <i>reg/mem8</i> , <i>imm8</i> | C0 /1 <i>ib</i> | Rotate an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| ROR <i>reg/mem16</i> , 1 | D1 /1 | Rotate a 16-bit register or memory location right once. |
| ROR <i>reg/mem16</i> , CL | D3 /1 | Rotate a 16-bit register or memory location right the number of bits specified in the CL register. |
| ROR <i>reg/mem16</i> , <i>imm8</i> | C1 /1 <i>ib</i> | Rotate a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| ROR <i>reg/mem32</i> , 1 | D1 /1 | Rotate a 32-bit register or memory location right once. |
| ROR <i>reg/mem32</i> , CL | D3 /13 | Rotate a 32-bit register or memory location right the number of bits specified in the CL register. |
| ROR <i>reg/mem32</i> , <i>imm8</i> | C1 /1 <i>ib</i> | Rotate a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value. |
| ROR <i>reg/mem64</i> , 1 | D1 /1 | Rotate a 64-bit register or memory location right once. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|--|
| ROR <i>reg/mem64, CL</i> | D3 /1 | Rotate a 64-bit register or memory operand right the number of bits specified in the CL register. |
| ROR <i>reg/mem64, imm8</i> | C1 /1 <i>ib</i> | Rotate a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value. |

Related Instructions

RCL, RCR, ROL, SHR, SHL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|----|----|----|----|-----|
| | | | | | | | | 1/0 | | | | | | | | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The source operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SAHF Store AH into Flags

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). The instruction ignores bits 1, 3, and 5 of register AH; it sets those bits in the EFLAGS register to 1, 0, and 0, respectively.

Using this instruction in 64-bit mode generates an invalid-opcode exception.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| SAHF | 9E | Loads the sign flag, the zero flag, the auxiliary flag, the parity flag, and the carry flag from the AH register into the lower 8 bits of the EFLAGS register. (Invalid in 64-bit mode.) |

Related Instructions

LAHF

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | | | X | Software was executing in 64-bit mode. |

SAL Shift Arithmetic Left

Shifts the bits of a register or memory location (first operand) to the left through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. For each bit shift, the SAL instruction clears the least-significant bit to 0. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A shift of 1 has a separate opcode encoding.

The effect of this instruction is multiplication by powers of two.

For 1-bit shifts, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is not affected.

If the shift count is 0, no flags are affected.

This instruction is synonymous with the SHL instruction.

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|---|
| <i>SAL reg/mem8, 1</i> | D0 /4 | Multiply an 8-bit register or memory location by two. |
| <i>SAL reg/mem8, CL</i> | D2 /4 | Multiply an 8-bit register or memory location by two the number of times specified in the CL register. |
| <i>SAL reg/mem8, imm8</i> | C0 /4 <i>ib</i> | Multiply an 8-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| <i>SAL reg/mem16, 1</i> | D1 /4 | Multiply a 16-bit register or memory location by two. |
| <i>SAL reg/mem16, CL</i> | D3 /4 | Multiply a 16-bit register or memory location by two the number of times specified in the CL register. |
| <i>SAL reg/mem16, imm8</i> | C1 /4 <i>ib</i> | Multiply a 16-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| <i>SAL reg/mem32, 1</i> | D1 /4 | Multiply a 32-bit register or memory location by two. |
| <i>SAL reg/mem32, CL</i> | D3 /4 | Multiply a 32-bit register or memory location by two the number of times specified in the CL register. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|---|
| <i>SAL reg/mem32, imm8</i> | <i>C1 /4 ib</i> | Multiply a 32-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| <i>SAL reg/mem64, 1</i> | <i>D1 /4</i> | Multiply a 64-bit register or memory location by two. |
| <i>SAL reg/mem64, CL</i> | <i>D3 /4</i> | Multiply a 64-bit register or memory location by two the number of times specified in the CL register. |
| <i>SAL reg/mem64, imm8</i> | <i>C1 /4 ib</i> | Multiply a 64-bit register or memory location by two the number of times specified by an 8-bit immediate value. |

Related Instructions

SAR, SHL, SHR, SHLD, SHRD

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory.. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SAR**Shift Arithmetic Right**

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The SAR instruction does not change the sign bit of the target operand. For each bit shift, it copies the sign bit to the next bit, preserving the sign of the result.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A shift of 1 has a separate opcode encoding.

For 1-bit shifts, the instruction clears the OF flag to 0. When the rotate count is greater than 1, the OF flag is not affected.

If the shift count is 0, no flags are affected.

Although the SAR instruction divides the operand by a power of 2, the behavior is different from the IDIV instruction. For example, shifting -11 (FFFFFFF5h) by two bits to the right (that is, divide -11 by 4), gives a result of FFFFFFFDh, or -3 , whereas the IDIV instruction for dividing -11 by 4 gives a result of -2 . This is because the IDIV instruction rounds off the quotient to zero, whereas the SAR instruction rounds off the remainder to zero for positive dividends and to negative infinity for negative dividends. So, for positive operands, SAR behaves like the corresponding IDIV instruction. For negative operands, it gives the same result if and only if all the shifted-out bits are zeroes; otherwise, the result is smaller by 1.

| Mnemonic | Opcode | Description |
|-----------------------------------|-----------------|--|
| SAR <i>reg/mem8</i> , 1 | D0 /7 | Divide a signed 8-bit register or memory operand by two. |
| SAR <i>reg/mem8</i> , CL | D2 /7 | Divide a signed 8-bit register or memory operand by two the number of times specified in the CL register. |
| SAR <i>reg/mem8</i> , <i>imm8</i> | C0 /7 <i>ib</i> | Divide a signed 8-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |
| SAR <i>reg/mem16</i> , 1 | D1 /7 | Divide a signed 16-bit register or memory operand by two. |

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|--|
| SAR <i>reg/mem16</i> , CL | D3 /7 | Divide a signed 16-bit register or memory operand by two the number of times specified in the CL register. |
| SAR <i>reg/mem16</i> , <i>imm8</i> | C1 /7 <i>ib</i> | Divide a signed 16-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |
| SAR <i>reg/mem32</i> , 1 | D1 /7 | Divide a signed 32-bit register or memory location by two. |
| SAR <i>reg/mem32</i> , CL | D3 /7 | Divide a signed 32-bit register or memory location by two the number of times specified in the CL register. |
| SAR <i>reg/mem32</i> , <i>imm8</i> | C1 /7 <i>ib</i> | Divide a signed 32-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| SAR <i>reg/mem64</i> , 1 | D1 /7 | Divide a signed 64-bit register or memory location by two. |
| SAR <i>reg/mem64</i> , CL | D3 /7 | Divide a signed 64-bit register or memory location by two the number of times specified in the CL register. |
| SAR <i>reg/mem64</i> , <i>imm8</i> | C1 /7 <i>ib</i> | Divide a signed 64-bit register or memory location by two the number of times specified by an 8-bit immediate value. |

Related Instructions

SAL, SHL, SHR, SHLD, SHRD

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------|------|-----------------|-----------|--|
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SBB**Subtract with Borrow**

Subtracts an immediate value or the value in a register or a memory location (second operand) from a register or a memory location (first operand), and stores the result in the first operand location. If the carry flag (CF) is 1, the instruction subtracts 1 from the result. Otherwise, it operates like SUB.

The operands cannot both be memory locations. The instruction sign-extends immediate value operands to the length of the first operand format.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

This instruction is useful for multibyte (multiword) numbers because it takes into account the borrow from a previous SUB instruction.

The memory-operand forms of the SBB instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|---|
| SBB AL, <i>imm8</i> | 1C <i>ib</i> | Subtract an immediate 8-bit value from the AL register with borrow. |
| SBB AX, <i>imm16</i> | 1D <i>iw</i> | Subtract an immediate 16-bit value from the AX register with borrow. |
| SBB EAX, <i>imm32</i> | 1D <i>id</i> | Subtract an immediate 32-bit value from the EAX register with borrow. |
| SBB RAX, <i>imm32</i> | 1D <i>id</i> | Subtract a sign-extended immediate 32-bit value from the RAX register with borrow. |
| SBB <i>reg/mem8</i> , <i>imm8</i> | 80 /3 <i>ib</i> | Subtract an immediate 8-bit value from an 8-bit register or memory location with borrow. |
| SBB <i>reg/mem16</i> , <i>imm16</i> | 81 /3 <i>iw</i> | Subtract an immediate 16-bit value from a 16-bit register or memory location with borrow. |
| SBB <i>reg/mem32</i> , <i>imm32</i> | 81 /3 <i>id</i> | Subtract an immediate 32-bit value from a 32-bit register or memory location with borrow. |
| SBB <i>reg/mem64</i> , <i>imm32</i> | 81 /3 <i>id</i> | Subtract an immediate 32-bit value from a 64-bit register or memory location with borrow. |

| Mnemonic | Opcode | Description |
|-----------------------------|-----------------|--|
| SBB <i>reg/mem16, imm8</i> | 83 /3 <i>ib</i> | Subtract a sign-extended 8-bit immediate value from a 16-bit register or memory location. |
| SBB <i>reg/mem32, imm8</i> | 83 /3 <i>ib</i> | Subtract a sign-extended 8-bit immediate value from a 32-bit register or memory location. |
| SBB <i>reg/mem64, imm8</i> | 83 /3 <i>ib</i> | Subtract a sign-extended 8-bit immediate value from a 64-bit register or memory location. |
| SBB <i>reg/mem8, reg8</i> | 18 / <i>r</i> | Subtract the contents of an 8-bit register from an 8-bit register or memory location. |
| SBB <i>reg/mem16, reg16</i> | 19 / <i>r</i> | Subtract the contents of a 16-bit register from a 16-bit register or memory location. |
| SBB <i>reg/mem32, reg32</i> | 19 / <i>r</i> | Subtract the contents of a 32-bit register from a 32-bit register or memory location. |
| SBB <i>reg/mem64, reg64</i> | 19 / <i>r</i> | Subtract the contents of a 64-bit register from a 64-bit register or memory location. |
| SBB <i>reg8, reg/mem8</i> | 1A / <i>r</i> | Subtract the contents of an 8-bit register or memory location from the contents of an 8-bit register. |
| SBB <i>reg16, reg/mem16</i> | 1B / <i>r</i> | Subtract the contents of a 16-bit register or memory location from the contents of an 16-bit register. |
| SBB <i>reg32, reg/mem32</i> | 1B / <i>r</i> | Subtract the contents of a 32-bit register or memory location from the contents of an 32-bit register. |
| SBB <i>reg64, reg/mem64</i> | 1B / <i>r</i> | Subtract the contents of a 64-bit register or memory location from the contents of an 64-bit register. |

Related Instructions

SUB, FSUB, SISUB

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SCAS Scan String

SCASB

SCASW

SCASD

SCASQ

Compares the byte, word, doubleword, or quadword pointed to by the rDI register with the value in the AL, AX, EAX, or RAX register, sets the status flags in the rFLAGS register according to the results, and then increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the DI register; otherwise, it decrements it. The instruction increments or decrements the DI register by the number of bytes in the operand.

The forms of the SCASx instruction with an explicit operand address the operand at ES:[rDI]. The explicit operand serves only to specify the type (size) of the values being compared.

The no-operands forms of the instruction use the ES:[rDI] registers to point to the value to be compared. The mnemonic determines the size of the operands and the specific register containing the other comparison value.

For block comparisons, the SCASx instructions support the REPE or REPZ prefixes (they are synonyms) and the REPNE or REPNZ prefixes (they are synonyms). For details about the REP prefixes, see “Repeat Prefixes” on page 10. A SCASx instruction can also operate inside a loop controlled by the LOOPcc instruction.

| Mnemonic | Opcode | Description |
|-------------------|--------|---|
| SCAS <i>mem8</i> | AE | Compare the contents of the AL register with the byte at DI, set status flags to reflect the outcome of the operation, and increment or decrement DI. |
| SCAS <i>mem16</i> | AF | Compare the contents of the AX register with the word at DI, set status flags to reflect the outcome of the operation, and increment or decrement DI. |
| SCAS <i>mem32</i> | AF | Compare the contents of the EAX register with the doubleword at EDI and set status flags to reflect the outcome of the operation, and increment or decrement EDI. |

| Mnemonic | Opcode | Description |
|-------------------|--------|--|
| SCAS <i>mem64</i> | AF | Compare the contents of the RAX register with the doubleword at RDI, set status flags to reflect the outcome of the operation, and increment or decrement RDI. |
| SCASB | AE | Compare the contents of the AL register with the byte at DI, set status flags to reflect the outcome of the operation, and increment or decrement DI. |
| SCASW | AF | Compare the contents of the AX register with the word at DI, set status flags to reflect the outcome of the operation, and increment or decrement DI. |
| SCASD | AF | Compare the contents of the EAX register with the doubleword at EDI, set status flags to reflect the outcome of the operation, and increment or decrement EDI. |
| SCASQ | AF | Compare the contents of the RAX register with the quadword at RDI, set status flags to reflect the outcome of the operation, and increment or decrement RDI. |

Related Instructions

SCASB, SCASD, SCASQ, SCASW

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|--------------|-----------|--|
| Stack, #SS | X | X | | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | During instruction execution, the effective address of the ES segment used for the operand, pointed to an illegal memory location. |
| | | | X | The ES register contained a null segment selector. |
| | | | X | The memory operand effective address in the ES segment was illegal. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| General protection, segment overrun, #GP | X | X | | A memory address exceeded a data segment limit or was non-canonical. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SETcc**Set Byte on Condition**

Checks the status flags in the rFLAGS register and, if the flags meet the condition specified in the mnemonic (*cc*), sets the value in the specified 8-bit memory location or register to 1.

Mnemonics with the A (above) and B (below) tags are intended for use when performing unsigned integer comparisons; those with G (greater) and L (lesser) tags are intended for use with signed integer comparisons.

Software typically uses the SETcc instructions to set logical indicators. Like the CMOVcc instructions (page 99), the SETcc instructions can replace two instructions—a conditional jump and a move. Replacing conditional jumps with conditional sets can help avoid branch-prediction penalties that may result from conditional jumps.

If the logical value “true” (logical one) is represented in a high-level language as an integer with all bits set to 1, software can accomplish such representation by first executing the opposite SETcc instruction—for example, the opposite of SETZ is SETNZ—and then decrementing the result.

| Mnemonic | Opcode | Description |
|---|---------------|---|
| SETA <i>reg/mem8</i> SETNB <i>reg/mem8</i> | 0F 97 | Set byte if the carry flag and the zero flag are both zero. |
| SETAE <i>reg/mem8</i> SETNB <i>reg/mem8</i> SETNC <i>reg/mem8</i> | 0F 93 | Set byte if the carry flag is zero. |
| SETB <i>reg/mem8</i> SETC <i>reg/mem8</i> SETNAE <i>reg/mem8</i> | 0F 92 | Set byte if the carry flag is 1. |
| SETBE <i>reg/mem8</i> SETNA <i>reg/mem8</i> | 0F 96 | Set byte if the carry flag is 1 or the zero flag is 1. |
| SETE <i>reg/mem8</i> SETZ <i>reg/mem8</i> | 0F 94 | Set byte if the zero flag is 1. |
| SETG <i>reg/mem8</i> SETNLE <i>reg/mem8</i> | 0F 9F | Set byte if the zero flag and the sign flag are equal to the overflow flag. |
| SETGE <i>reg/mem8</i> SETNL <i>reg/mem8</i> | 0F 9D | Set byte if the sign flag is equal to the overflow flag. |
| SETL <i>reg/mem8</i> SETNGE <i>reg/mem8</i> | 0F 9C | Set byte if the sign flag is not equal to the overflow flag. |

| Mnemonic | Opcode | Description |
|--|---------------|--|
| SETLE <i>reg/mem8</i> SETNG <i>reg/mem8</i> | 0F 9E | Set byte if the zero flag is 1 or the sign flag and the overflow flag are not equal. |
| SETNE <i>reg/mem8</i> SETNZ <i>reg/mem8</i> | 0F 95 | Set byte if the zero flag is zero. |
| SETNO <i>reg/mem8</i> | 0F 91 | Set byte if the overflow flag is zero. |
| SETNP <i>reg/mem8</i> SETPO <i>reg/mem8</i> | 0F 9B | Set byte if the parity flag is zero. |
| SETNS <i>reg/mem8</i> | 0F 99 | Set byte if the sign flag is zero. |
| SETO <i>reg/mem8</i> | 0F 90 | Set byte if the overflow flag is 1. |
| SETP <i>reg/mem8</i> SETPE <i>reg/mem8</i> | 0F 9A | Set byte if the parity flag is 1. |
| SETS <i>reg/mem8</i> | 0F 98 | Set byte if the sign flag is 1. |

Related Instructions

None

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | | An unaligned-memory data reference was performed while alignment checking was enabled. |

SFENCE Store Fence

Acts as a barrier to force strong memory ordering (serialization) between store instructions preceding the SFENCE and store instructions that follow the SFENCE. A weakly-ordered memory system allows hardware to reorder reads and writes between the processor and memory. The SFENCE instruction guarantees that the system completes all previous stores before executing subsequent stores.

The SFENCE instruction is weakly-ordered with respect to load instructions, data and instruction prefetches, and the LFENCE instruction. Speculative loads initiated by the processor, or specified explicitly using cache-prefetch instructions, can be reordered around an SFENCE.

In addition to store instructions, SFENCE is strongly ordered with respect to other SFENCE instructions, MFENCE instructions, and serializing instructions.

Support for the SFENCE instruction is indicated when the SSE bit (bit 25) is set to 1 in EDX after executing CUID with extended function 8000_0001h.

| Mnemonic | Opcode | Description |
|----------|----------|---|
| SFENCE | 0F AE F8 | Force strong ordering of (serialized) store operations. |

Related Instructions

LFENCE, MFENCE

rFLAGS Affected

None

Exceptions

None

SHL Shift Left

Shifts the bits of a register or memory location (first operand) to the left through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. For each bit shift, the SHL instruction clears the least-significant bit to 0. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A shift of 1 has a separate opcode encoding.

The effect of this instruction is multiplication by powers of two.

For 1-bit shifts, the instruction sets the OF flag to the exclusive OR of the CF bit (after the rotate) and the most significant bit of the result. When the rotate count is greater than 1, the OF flag is not affected.

If the shift count is 0, no flags are affected.

This instruction is synonymous with the SAL instruction.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| SHL <i>reg/mem8</i> , 1 | D0 /4 | Multiply an 8-bit register or memory location by two. |
| SHL <i>reg/mem8</i> , CL | D2 /4 | Multiply an 8-bit register or memory location by two the number of times specified in the CL register. |
| SHL <i>reg/mem8</i> , <i>imm8</i> | C0 /4 <i>ib</i> | Multiply an 8-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| SHL <i>reg/mem16</i> , 1 | D1 /4 | Multiply a 16-bit register or memory location by two. |
| SHL <i>reg/mem16</i> , CL | D3 /4 | Multiply a 16-bit register or memory location by two the number of times specified in the CL register. |
| SHL <i>reg/mem16</i> , <i>imm8</i> | C1 /4 <i>ib</i> | Multiply a 16-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| SHL <i>reg/mem32</i> , 1 | D1 /4 | Multiply a 32-bit register or memory location by two. |
| SHL <i>reg/mem32</i> , CL | D3 /4 | Multiply a 32-bit register or memory location by two the number of times specified in the CL register. |

| Mnemonic | Opcode | Description |
|----------------------------|-----------------|---|
| SHL <i>reg/mem32, imm8</i> | C1 /4 <i>ib</i> | Multiply a 32-bit register or memory location by two the number of times specified by an 8-bit immediate value. |
| SHL <i>reg/mem64, 1</i> | D1 /4 | Multiply a 64-bit register or memory location by two. |
| SHL <i>reg/mem64, CL</i> | D3 /4 | Multiply a 64-bit register or memory location by two the number of times specified in the CL register. |
| SHL <i>reg/mem64, imm8</i> | C1 /4 <i>ib</i> | Multiply a 64-bit register or memory location by two the number of times specified by an 8-bit immediate value. |

Related Instructions

SHR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SHLD Shift Left Double

Shifts the bits of a register or memory location (first operand) to the left by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the right. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. If the count specified in CL is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are affected.

If the count is 1 and the sign of the value being shifted changes, the instruction sets the OF flag to 1.

| Mnemonic | Opcode | Description |
|------------------------------------|--------|---|
| SHLD <i>reg/mem16, reg16, imm8</i> | 0F A4 | Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value and shift bits out of a 16-bit register into the destination operand from the right. |
| SHLD <i>reg/mem16, reg16, CL</i> | 0F A5 | Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in the CL register and shift bits out of a 16-bit register into the destination operand from the right. |
| SHLD <i>reg/mem32, reg32, imm8</i> | 0F A4 | Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value and shift bits out of a 32-bit register into the destination operand from the right. |
| SHLD <i>reg/mem32, reg32, CL</i> | 0F A5 | Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in the CL register and shift bits out of a 32-bit register into the destination operand from the right. |

| | | |
|------------------------------------|-------|---|
| SHLD <i>reg/mem64, reg64, imm8</i> | 0F A4 | Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value and shift bits out of a 64-bit register into the destination operand from the right. |
| SHLD <i>reg/mem64, reg64, CL</i> | 0F A5 | Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in the CL register and shift bits out of a 64-bit register into the destination operand from the right. |

Action

```

cnt = mod(COUNT,64);
sz = Sizeof(DEST);
IF (cnt == 0) (NOP)
else {
    if (cnt >= sz) then
    {
        // bad parameters
        // DEST is undefined;
        // CF, OF, SF, ZF, AF, PF are undefined;
    }else{ // Perform the shift
        CF = BIT[DEST, cnt - 1]; (* last bit shifted out on exit *)
        for (i = (sz - 1); i >= cnt; i--) DEST[i] = DEST[i-cnt];
        for (i = cnt-1; i >= 0; i--) DEST[i] = inBits[i-cnt+sz];
    }
}

```

Related Instructions

SHRD, SAL, SAR, SHR, SHL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13-12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31-22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SHR Shift Right

Shifts the bits of a register or memory location (first operand) to the right through the CF bit by the number of bit positions in an unsigned immediate value or the CL register (second operand). The instruction discards bits shifted out of the CF flag. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

For each bit shift, the instruction clears the most-significant bit to 0.

The effect of this instruction is unsigned division by powers of two.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. A shift of 1 has a separate opcode encoding.

For 1-bit shifts, the instruction sets the OF flag to the most-significant bit of the original value. When the rotate count is greater than 1, the OF flag is not affected.

If the shift count is 0, no flags are affected.

| Mnemonic | Opcode | Description |
|------------------------------------|-----------------|---|
| SHR <i>reg/mem8</i> , 1 | D0 /5 | Unsigned divide an 8-bit register or memory operand by two. |
| SHR <i>reg/mem8</i> , CL | D2 /5 | Unsigned divide an 8-bit register or memory operand by two the number of times specified in the CL register. |
| SHR <i>reg/mem8</i> , <i>imm8</i> | C0 /5 <i>ib</i> | Unsigned divide an 8-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |
| SHR <i>reg/mem16</i> , 1 | D1 /5 | Unsigned divide a 16-bit register or memory operand by two. |
| SHR <i>reg/mem16</i> , CL | D3 /5 | Unsigned divide a 16-bit register or memory operand by two the number of times specified in the CL register. |
| SHR <i>reg/mem16</i> , <i>imm8</i> | C1 /5 <i>ib</i> | Unsigned divide a 16-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |
| SHR <i>reg/mem32</i> , 1 | D1 /5 | Unsigned divide a 32-bit register or memory operand by two. |
| SHR <i>reg/mem32</i> , CL | D3 /5 | Unsigned divide a 32-bit register or memory operand by two the number of times specified in the CL register. |
| SHR <i>reg/mem32</i> , <i>imm8</i> | C1 /5 <i>ib</i> | Unsigned divide a 32-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |

| | | |
|------------------------------------|-----------------|---|
| SHR <i>reg/mem64</i> , 1 | D1 /5 | Unsigned divide a 64-bit register or memory operand by two. |
| SHR <i>reg/mem64</i> , CL | D3 /5 | Unsigned divide a 64-bit register or memory operand by two the number of times specified in the CL register. |
| SHR <i>reg/mem64</i> , <i>imm8</i> | C1 /5 <i>ib</i> | Unsigned divide a 64-bit register or memory operand by two the number of times specified by an 8-bit immediate value. |

Related Instructions

SHL, SHR, SAL, SAR, SHLD, SHRD

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
 Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory.. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SHRD Shift Right Double

Shifts the bits of a register or memory location (first operand) to the right by the number of bit positions in an unsigned immediate value or the CL register (third operand), and shifts in a bit pattern (second operand) from the left. At the end of the shift operation, the CF flag contains the last bit shifted out of the first operand.

The processor masks the upper three bits of the count operand, thus restricting the count to a number between 0 and 31. In 64-bit mode, it masks the upper two bits of the count, providing a count in the range of 0 to 63. If the count specified in CL is greater than the operand size, the result in the destination register is undefined.

If the shift count is 0, no flags are affected.

If the count is 1 and the sign of the value being shifted changes, the instruction sets the OF flag to 1.

| Mnemonic | Opcode | Description |
|------------------------------------|--------|---|
| SHRD <i>reg/mem16, reg16, imm8</i> | 0F AC | Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value and shift bits out of a 16-bit register into the destination operand from the left. |
| SHRD <i>reg/mem16, reg16, CL</i> | 0F AD | Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in the CL register and shift bits out of a 16-bit register into the destination operand from the left. |
| SHRD <i>reg/mem32, reg32, imm8</i> | 0F AC | Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value and shift bits out of a 32-bit register into the destination operand from the left. |
| SHRD <i>reg/mem32, reg32, CL</i> | 0F AD | Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in the CL register and shift bits out of a 32-bit register into the destination operand from the left. |

| Mnemonic | Opcode | Description |
|------------------------------------|--------|---|
| SHRD <i>reg/mem16, reg64, imm8</i> | 0F AC | Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value and shift bits out of a 64-bit register into the destination operand from the left. |
| SHRD <i>reg/mem64, reg64, CL</i> | 0F AD | Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in the CL register and shift bits out of a 64-bit register into the destination operand from the left. |

Action

```

cnt = mod(COUNT,64);
sz = Sizeof(DEST);
IF (cnt == 0) (NOP)
else {
    if (cnt >= sz) then
    {
        // bad parameters
        // DEST is undefined;
        // CF, OF, SF, ZF, AF, PF are undefined;
    }else{ (* Perform the shift *)
        CF = BIT[DEST, cnt - 1]; (* last bit shifted out on exit *)
        for (i = 0; i < (sz - cnt - 1); i++) DEST[i - cnt] = DEST[i];
        for (i = (sz - cnt); i < sz - 1; i++) DEST[i] = inBits[i+cnt-sz];
    }
}

```

Related Instructions

SHLD, SAL, SAR, SHR, SHL

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13-12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31-22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

STC Set Carry Flag

Sets the carry flag (CF) to 1 in the rFLAGS register.

The OF, ZF, SF, AF, DF, and PF flags are unaffected.

| Mnemonic | Opcode | Description |
|----------|--------|-----------------|
| STC | F9 | Set carry flag. |

Related Instructions

CLC

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | | | | | | | | 1 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

None

STD Set Direction Flag

Sets DF to 1 in the rFLAGS register. Used widely with string instructions. If the DF flag is 1 during the execution of a string instruction, the instruction decrements the string pointers are decremented. If the DF flag is 0, it increments the string pointers.

With the exception of I/O string instructions, all string operations use rSI as the source-operand pointer and rDI as the destination-operand pointer. I/O string instructions use rDX to specify the input-port or output-port number. Repeated string operations (those preceded with a repeat-instruction prefix) increment or decrement the rSI and rDI registers as they move string elements from the source location to the destination. Repeat-string operations also use rCX to hold the string length, and decrement it as they move data from one location to the other.

The CF, OF, ZF, SF, AF, and PF flags are unaffected.

| Mnemonic | Opcode | Description |
|----------|--------|-------------------------|
| STD | FD | Set the direction flag. |

Related Instructions

CLD, INS_x, LODS_x, MOVS_x, OUTS_x, SCAS_x, STOS_x

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | 1 | | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

None

STOS **Store String**

STOSB

STOSW

STOSD

STOSQ

Copies a byte, word, doubleword, or quadword from the AL, AX, EAX, or RAX registers to the memory location pointed to by ES:rDI and increments or decrements the rDI register according to the state of the DF flag in the rFLAGS register.

If the DF flag is 0, the instruction increments the pointer; otherwise, it decrements the pointer. It increments or decrements the pointer by 1, 2, 4, or 8, depending on the size of the value being copied.

The forms of the STOSx instruction with an explicit operand use the operand only to specify the type (size) of the value being copied.

The no-operands forms specify the type (size) of the value being copied with the mnemonic.

The STOSx instructions support the REP prefixes. For details about the REP prefixes, see “Repeat Prefixes” on page 10. The STOSx instructions can also operate inside a LOOPcc instruction.

| Mnemonic | Opcode | Description |
|-------------------|---------------|--|
| STOS <i>reg8</i> | AA | Store the contents of the AL register to DI. |
| STOS <i>reg16</i> | AB | Store the contents of the AX register to DI. |
| STOS <i>reg32</i> | AB | Store the contents of the EAX register to EDI. |
| STOS <i>reg64</i> | AB | Store the contents of the RAX register to RDI. |
| STOSB | AA | Store the contents of the AL register to DI. |
| STOSW | AB | Store the contents of the AX register to DI. |
| STOSD | AB | Store the contents of the EAX register to EDI. |
| STOSQ | AB | Store the contents of the RAX register to RDI. |

Related Instructions

LEA, LDS, LES, LFS, LGS, LSS

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | The memory operand effective address exceeded the limit of the ES segment. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | The ES register contained a null segment selector. |
| General protection, segment overrun, #GP | X | X | | The memory operand effective address exceeded the limit of the ES segment. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SUB Subtract

Subtracts an immediate value or the value in a register or memory location (second operand) from a register or a memory location (first operand) and stores the result in the first operand location. An immediate value is sign-extended to the length of the first operand.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a borrow in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

The memory-operand forms of the SUB instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|-------------------------------------|-----------------|---|
| SUB AL, <i>imm8</i> | 2C <i>ib</i> | Subtract an immediate 8-bit value from the AL register and store the result in AL. |
| SUB AX, <i>imm16</i> | 2D <i>iv</i> | Subtract an immediate 16-bit value from the AX register and store the result in AX. |
| SUB EAX, <i>imm32</i> | 2D <i>id</i> | Subtract an immediate 32-bit value from the EAX register and store the result in EAX. |
| SUB RAX, <i>imm32</i> | 2D <i>id</i> | Subtract a sign-extended immediate 32-bit value from the RAX register and store the result in RAX. |
| SUB <i>reg/mem8</i> , <i>imm8</i> | 80 /5 <i>ib</i> | Subtract an immediate 8-bit value from an 8-bit destination register or memory location. |
| SUB <i>reg/mem16</i> , <i>imm16</i> | 81 /5 <i>iv</i> | Subtract an immediate 16-bit value from an 16-bit destination register or memory location. |
| SUB <i>reg/mem32</i> , <i>imm32</i> | 81 /5 <i>id</i> | Subtract an immediate 32-bit value from an 32-bit destination register or memory location. |
| SUB <i>reg/mem64</i> , <i>imm32</i> | 81 /5 <i>id</i> | Subtract a sign-extended immediate 32-bit value from an 64-bit destination register or memory location. |
| SUB <i>reg/mem16</i> , <i>imm8</i> | 83 /5 <i>ib</i> | Subtract a sign-extended immediate 8-bit value from a 16-bit register or memory location. |
| SUB <i>reg/mem32</i> , <i>imm8</i> | 83 /5 <i>ib</i> | Subtract a sign-extended immediate 8-bit value from a 32-bit register or memory location. |
| SUB <i>reg/mem64</i> , <i>imm8</i> | 83 /5 <i>ib</i> | Subtract a sign-extended immediate 8-bit value from a 64-bit register or memory location. |

| Mnemonic | Opcode | Description |
|-----------------------------|--------|---|
| SUB <i>reg/mem8, reg8</i> | 28 /r | Subtract the contents of an 8-bit register from an 8-bit destination register or memory location. |
| SUB <i>reg/mem16, reg16</i> | 29 /r | Subtract the contents of a 16-bit register from a 16-bit destination register or memory location. |
| SUB <i>reg/mem32, reg32</i> | 29 /r | Subtract the contents of a 32-bit register from a 32-bit destination register or memory location. |
| SUB <i>reg/mem64, reg64</i> | 29 /r | Subtract the contents of a 64-bit register from a 64-bit destination register or memory location. |
| SUB <i>reg8, reg/mem8</i> | 2A /r | Subtract the contents of an 8-bit register or memory operand from an 8-bit destination register. |
| SUB <i>reg16, reg/mem16</i> | 2B /r | Subtract the contents of a 16-bit register or memory operand from a 16-bit destination register. |
| SUB <i>reg32, reg/mem32</i> | 2B /r | Subtract the contents of a 32-bit register or memory operand from a 32-bit destination register. |
| SUB <i>reg64, reg/mem64</i> | 2B /r | Subtract the contents of a 64-bit register or memory operand from a 64-bit destination register. |

Related Instructions

ADC, ADD

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|---|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking is enabled. |

TEST**Test Bits**

Performs a bit-wise logical AND on the value in a register or memory location (first operand) with an immediate value or the value in a register (second operand) and sets the flags in the rFLAGS register based on the result. While the AND instruction changes the contents of the destination and the flag bits, the TEST instruction changes only the flag bits as shown in the **Action** section.

| Mnemonic | Opcode | Description |
|--------------------------------------|-----------------|--|
| TEST AL, <i>imm8</i> | A8 <i>ib</i> | AND an immediate 8-bit value with the contents of the AL register and set rFLAGS to reflect the result. |
| TEST AX, <i>imm16</i> | A9 <i>iv</i> | AND an immediate 16-bit value with the contents of the AX register and set rFLAGS to reflect the result. |
| TEST EAX, <i>imm32</i> | A9 <i>id</i> | AND an immediate 32-bit value with the contents of the EAX register and set rFLAGS to reflect the result. |
| TEST RAX, <i>imm32</i> | A9 <i>id</i> | AND an immediate 32-bit value with the contents of the RAX register and set rFLAGS to reflect the result. |
| TEST <i>reg/mem8</i> , <i>imm8</i> | F6 /0 <i>ib</i> | AND an immediate 8-bit value with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result. |
| TEST <i>reg/mem16</i> , <i>imm16</i> | F7 /0 <i>iv</i> | AND an immediate 32-bit value with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result. |
| TEST <i>reg/mem32</i> , <i>imm32</i> | F7 /0 <i>id</i> | AND an immediate 32-bit value with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result. |
| TEST <i>reg/mem64</i> , <i>imm32</i> | F7 /0 <i>id</i> | AND an immediate 32-bit value with the contents of a 64-bit register or memory operand and set the SF, ZF, and PF flags to reflect the result. |
| TEST <i>reg/mem8</i> , <i>reg8</i> | 84 / <i>r</i> | AND the contents of an 8-bit register with the contents of an 8-bit register or memory operand and set the SF, ZF, and PF flags to reflect the result. |
| TEST <i>reg/mem16</i> , <i>reg16</i> | 85 / <i>r</i> | AND the contents of a 16-bit register with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result. |
| TEST <i>reg/mem32</i> , <i>reg32</i> | 85 / <i>r</i> | AND the contents of a 32-bit register with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result. |
| TEST <i>reg/mem64</i> , <i>reg64</i> | 85 / <i>r</i> | AND the contents of a 64-bit register with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result. |

Action

```

for (i = 0; i < sizeof (src1); i++) tmp[i] = src1[i] && src2[i];
SF = msb(tmp);
if (tmp == 0) ZF = 0;
else ZF = 1;
PF = BitwiseXNOR(tmp[0:7]);
CF = 0;
OF = 0;

```

Related Instructions

AND, CMP, SUB

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|-----|----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | 1/0 | U | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

XADD Exchange and Add

Exchanges the contents of a register (second operand) with the contents of a register or memory location (first operand), computes the sum of the two values, and stores the result in the first operand location.

The memory-operand forms of the XADD instruction support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

| Mnemonic | Opcode | Description |
|------------------------------|----------|---|
| XADD <i>reg/mem8, reg8</i> | 0F C0 /r | Exchange the contents of an 8-bit register and an 8-bit destination register or memory operand and load their sum into the destination. |
| XADD <i>reg/mem16, reg16</i> | 0F C1 /r | Exchange the contents of a 16-bit register and a 16-bit destination register or memory operand and load their sum into the destination. |
| XADD <i>reg/mem32, reg32</i> | 0F C1 /r | Exchange the contents of a 32-bit register and a 32-bit destination register or memory operand and load their sum into the destination. |
| XADD <i>reg/mem64, reg64</i> | 0F C1 /r | Exchange the contents of a 64-bit register and a 64-bit destination register or memory operand and load their sum into the destination. |

Related Instructions

ADD, XCHG

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|-----|----|----|----|-----|----|-----|-----|-----|
| | | | | | | | | 1/0 | | | | 1/0 | | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was being used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

XCHG Exchange

Exchanges the contents of the two operands. The operands can be two general-purpose registers or a register and a memory location. If either operand references memory, the processor locks automatically, whether or not the LOCK prefix is used and independently of the value of IOPL. For details about the LOCK prefix, see “Lock Prefix” on page 10.

The x86 architecture commonly uses the XCHG EAX, EAX instruction (opcode 90h) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h as a true NOP only if it would exchange rAX with itself. Opcode 90h can still be used to exchange RAX and another 64-bit register (e.g., R8) if the appropriate REX prefix is used. Without this special handling, the instruction would zero-extend the upper 32 bits of RAX, and thus it would not be a true no-operation.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction.

| Mnemonic | Opcode | Description |
|--------------------------------------|----------------|--|
| XCHG AX, <i>reg16</i> | 90 + <i>rw</i> | Exchange the contents of the AX register with the contents of a 16-bit register. |
| XCHG <i>reg16</i> , AX | 90 + <i>rw</i> | Exchange the contents of a 16-bit register with the contents of the AX register. |
| XCHG EAX, <i>reg32</i> | 90 + <i>rd</i> | Exchange the contents of the EAX register with the contents of a 32-bit register. |
| XCHG <i>reg32</i> , EAX | 90 + <i>rd</i> | Exchange the contents of a 32-bit register with the contents of the EAX register. |
| XCHG RAX, <i>reg64</i> | 90 + <i>rq</i> | Exchange the contents of the RAX register with the contents of a 64-bit register. |
| XCHG <i>reg64</i> , RAX | 90 + <i>rq</i> | Exchange the contents of a 64-bit register with the contents of the RAX register. |
| XCHG <i>reg/mem8</i> , <i>reg8</i> | 86 / <i>r</i> | Exchange the contents of an 8-bit register with the contents of an 8-bit register or memory operand. |
| XCHG <i>reg8</i> , <i>reg/mem8</i> | 86 / <i>r</i> | Exchange the contents of an 8-bit register or memory operand with the contents of an 8-bit register. |
| XCHG <i>reg/mem16</i> , <i>reg16</i> | 87 / <i>r</i> | Exchange the contents of a 16-bit register with the contents of a 16-bit register or memory operand. |

| Mnemonic | Opcode | Description |
|------------------------------|---------------|--|
| XCHG <i>reg16, reg/mem16</i> | 87 /r | Exchange the contents of a 16-bit register or memory operand with the contents of a 16-bit register. |
| XCHG <i>reg/mem32, reg32</i> | 87 /r | Exchange the contents of a 32-bit register with the contents of a 32-bit register or memory operand. |
| XCHG <i>reg32, reg/mem32</i> | 87 /r | Exchange the contents of a 32-bit register or memory operand with the contents of a 32-bit register. |
| XCHG <i>reg/mem64, reg64</i> | 87 /r | Exchange the contents of a 64-bit register with the contents of a 64-bit register or memory operand. |
| XCHG <i>reg64, reg/mem64</i> | 87 /r | Exchange the contents of a 64-bit register or memory operand with the contents of a 64-bit register. |

Related Instructions

BSWAP

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

XLAT

XLATB

Translate Table Index

Uses the unsigned integer in the AL register as an offset into a table and copies the contents of the table entry at that location to the AL register.

The instruction uses *seg:[rSI]* as the base address of the table. The value of *seg* defaults to the DS segment, but may be overridden by a segment prefix.

This instruction writes AL without changing RAX[63:8]. This instruction ignores operand size.

The single-operand form of the XLAT instruction uses the operand to document the segment and address size attribute, but it uses the base address specified by the rBX registers.

This instruction is often used to translate data from one format (such as ASCII) to another (such as EBCDIC).

| Mnemonic | Opcode | Description |
|------------------|---------------|---|
| XLAT <i>mem8</i> | D7 | Set AL to the contents of DS:[rBX + unsigned AL]. |
| XLATB | D7 | Set AL to the contents of DS:[rBX + unsigned AL]. |

Related Instructions

XLATB

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|-------------|-------------------------|------------------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

XOR Logical Exclusive OR

Performs a bitwise exclusive OR operation on both operands and stores the result in the first operand location. The first operand can be a register or memory location. The second operand can be an immediate value, a register, or a memory location. The instruction `XOR AX, AX` clears `AX`.

The forms of the XOR instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see “Lock Prefix” on page 10.

The instruction performs the following operation for each bit:

| X | Y | X XOR Y |
|---|---|---------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| Mnemonic | Opcode | Description |
|-----------------------------------|-----------------------|--|
| <code>XOR AL, imm8</code> | <code>34 ib</code> | XOR the contents of AL with an immediate 8-bit operand and store the result in AL. |
| <code>XOR AX, imm16</code> | <code>35 iw</code> | XOR the contents of AX with an immediate 16-bit operand and store the result in AX. |
| <code>XOR EAX, imm32</code> | <code>35 id</code> | XOR the contents of EAX with an immediate 32-bit operand and store the result in EAX. |
| <code>XOR RAX, imm32</code> | <code>35 id</code> | XOR the contents of RAX with a sign-extended immediate 32-bit operand and store the result in RAX. |
| <code>XOR reg/mem8, imm8</code> | <code>80 /6 ib</code> | XOR the contents of an 8-bit destination register or memory operand with an 8-bit immediate value and store the result in the destination. |
| <code>XOR reg/mem16, imm16</code> | <code>81 /6 iw</code> | XOR the contents of a 16-bit destination register or memory operand with a 16-bit immediate value and store the result in the destination. |

| Mnemonic | Opcode | Description |
|-----------------------------|-----------------|---|
| <i>XOR reg/mem32, imm32</i> | 81 /6 <i>id</i> | XOR the contents of a 32-bit destination register or memory operand with a 32-bit immediate value and store the result in the destination. |
| <i>XOR reg/mem64, imm32</i> | 81 /6 <i>id</i> | XOR the contents of a 64-bit destination register or memory operand with a 32-bit immediate value and store the result in the destination. |
| <i>XOR reg/mem16, imm8</i> | 83 /6 <i>ib</i> | XOR the contents of a 16-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination. |
| <i>XOR reg/mem32, imm8</i> | 83 /6 <i>ib</i> | XOR the contents of a 32-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination. |
| <i>XOR reg/mem64, imm8</i> | 83 /6 <i>ib</i> | XOR the contents of a 64-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination. |
| <i>XOR reg/mem8, reg8</i> | 30 / <i>r</i> | XOR the contents of an 8-bit destination register with the contents of an 8-bit register or memory operand and store the result in the destination. |
| <i>XOR reg/mem16, reg16</i> | 31 / <i>r</i> | XOR the contents of a 16-bit destination register with the contents of a 16-bit register or memory operand and store the result in the destination. |
| <i>XOR reg/mem32, reg32</i> | 31 / <i>r</i> | XOR the contents of a 32-bit destination register with the contents of a 32-bit register or memory operand and store the result in the destination. |
| <i>XOR reg/mem64, reg64</i> | 31 / <i>r</i> | XOR the contents of a 64-bit destination register with the contents of a 64-bit register or memory operand and store the result in the destination. |
| <i>XOR reg8, reg/mem8</i> | 32 / <i>r</i> | XOR the contents of an 8-bit destination register with the contents of an 8-bit register or memory operand and store the results in the destination. |
| <i>XOR reg16, reg/mem16</i> | 33 / <i>r</i> | XOR the contents of a 16-bit destination register with the contents of a 16-bit register or memory operand and store the results in the destination. |
| <i>XOR reg32, reg/mem32</i> | 33 / <i>r</i> | XOR the contents of a 32-bit destination register with the contents of a 32-bit register or memory operand and store the results in the destination. |
| <i>XOR reg64, reg/mem64</i> | 33 / <i>r</i> | XOR the contents of a 64-bit destination register with the contents of a 64-bit register or memory operand and store the results in the destination. |

Related Instructions

OR, AND, NOT

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|-----|-----|----|-----|----|
| | | | | | | | | 0 | | | | 1/0 | 1/0 | U | 1/0 | 0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
 Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a non-writable segment. |
| | | | X | A null data segment was used to reference memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

4 System Instruction Reference

This chapter describes the function, mnemonic syntax, opcodes, affected flags, and possible exceptions generated by the system instructions. The system instructions are used to establish the operating mode, access processor resources, handle program and system errors, and manage memory. Many of these instructions can only be executed by software, such as the operating system kernel and interrupt handlers, which runs at the highest privilege level. Only system instructions can access certain processor resources, such as the control registers, model-specific registers, and debug registers.

System instructions are supported in all hardware implementations of the x86-64 architecture, except that the following system instructions are implemented only if their associated CPUID function bits are set:

- SYSENTER and SYSEXIT, indicated by bit 11 of CPUID standard function 1.
- SYSCALL and SYSRET, indicated by bit 11 of CPUID extended function 8000_0001h.
- Long Mode instructions, indicated by bit 29 of CPUID extended function 8000_0001h.
- RDMSR and WRMSR, indicated by bit 5 of CPUID standard function 1 or extended function 8000_0001h.

There are also several other CPUID function bits that control the use of system resources and functions, such as paging functions, virtual-mode extensions, machine-check exceptions, advanced programmable interrupt control (APIC), memory-type range registers (MTRRs), etc. For details, see “Processor Feature Identification” in Volume 2.

For further information about the system instructions and register resources, see:

- “System-Management Instructions” in Volume 2.
- “Summary of Registers and Data Types” on page 28.
- “Notation” on page 40.
- “Instruction Prefixes” on page 3.

ARPL

Adjust Requestor Privilege Level

The ARPL instruction compares the requestor privilege level (RPL) fields of two segment selectors in the source and destination operands of the instruction. The destination operand can be either a 16-bit register or memory location; the source operand must be a 16-bit register. If the RPL field of the destination operand is less than the RPL field of the segment selector in the source register, then the zero flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the destination operand remains unchanged and the zero flag is cleared.

| Mnemonic | Opcode | Description |
|------------------------------|--------|--|
| ARPL <i>reg/mem16, reg16</i> | 63 /r | Adjust the requested privilege level of a destination 16-bit register or memory operand to a level not less than the requested privilege level of the segment selector specified in the 16-bit source register. (Invalid in 64-bit mode.) |

The ARPL instruction is intended for use by operating-system procedures to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. The segment selector passed to the operating system is placed in the destination operand and the segment selector for the code segment of the application program is placed in the source operand. The RPL field in the source operand represents the privilege level of the application program. The ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower than the privilege level of the application program.

See “Adjusting Access Rights” in Volume 2, for more information on access rights.

In 64-bit mode, this opcode (63H) is used for the MOVSSXD instruction.

Related Instructions

LAR, LSL, VERR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | | | | | | 1/0 | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | Result was located in a nonwritable segment. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

CLI Clear Interrupt Flag

Resets the interrupt flag (IF) in the rFLAGS register to zero, thereby masking external interrupts received on the INTR input. Interrupts received on the nonmaskable interrupt (NMI) input are not blocked by this instruction.

| Mnemonic | Opcode | Description |
|----------|--------|-----------------------------------|
| CLI | FA | Clear the interrupt flag to zero. |

In protected mode and virtual-8086-mode, this instruction is IOPL-sensitive. The processor takes a general-protection exception (#GP) whenever an IOPL-sensitive instruction is executed and the rFLAGS.IOPL field is less than the CPL. Because all virtual-8086 programs run at CPL = 3, system software can interrupt all instructions that modify the IF bit by setting the IOPL less than 3.

In virtual-8086 mode, if the virtual-8086-mode extensions are enabled (CR4.VME = 1) or if the protected mode virtual interrupt (PVI) flag is set in protected mode (CR4.VPI = 1), then the CLI instruction also clears the rFlags.VIF flag.

See “Virtual-8086 Mode Extensions” in Volume 2 for more information about IOPL-sensitive instructions.

Action

```
if (CPL <= IOPL)
    EFLAGS.IF = 0;

else if ((mode==virtual) && CR4.VME) || ((mode==protected)&&CR4.PVI) && (CPL ==3)
    EFLAGS.VIF = 0;

else
    #GP
```

Related Instructions

STI

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | 0 | | | | | | | | 0 | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--|
| General protection, #GP | | X | X | The CPL was greater than the IOPL of the current task. |

CLTS**Clear Task-Switched Flag in CR0**

Clears the task-switched (TS) flag in the CR0 register to 0. The processor sets the TS flag on each task switch. The CLTS instruction is intended to facilitate the synchronization of FPU context saves during multitasking operations.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| CLTS | 0F 06 | Clears the task-switched flag in control register 0. |

This instruction can only be used if the current privilege level is 0.

See “System-Control Registers” in Volume 2 for more information on FPU synchronization and the TS flag.

Action

```
taskSwitchedFlag = 0;
```

Related Instructions

MOV CR_n

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--------------------|
| General protection, #GP | | X | X | CPL was not 0. |

HLT**Halt**

Causes the microprocessor to halt instruction execution and enter the HALT state. Execution resumes when an unmasked hardware interrupt (INTR), non-maskable interrupt (NMI), system management interrupt (SMI), RESET or INIT occurs. Entering the halt state puts the processor in low-power mode.

| Mnemonic | Opcode | Description |
|----------|--------|------------------------------|
| HLT | F4 | Halts instruction execution. |

If an INTR or NMI is used to resume execution after a HLT instruction, the saved instruction pointer points to the instruction following the HLT instruction.

Before executing a HLT instruction, you should enable hardware interrupts (with the STI instruction). Do not clear the interrupt flag (IF) with the CLI instruction before executing a HLT instruction. Doing so disables INTRs, and the system will remain in a halt state until an NMI, SMI, RESET or INIT occurs.

If an SMI brings the processor out of the halt state, the SMI handler can decide whether to return to the HALT state or not. See Volume 2, *System Programming*, for information on SMIs.

Program privilege level must be 0 to execute this instruction.

Related Instructions

STI, CLI

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--------------------|
| General protection, #GP | | X | X | CPL was not 0. |

INT 3 Interrupt to Debug Vector

Calls the debug exception handler. This is the one-byte version of the two-byte INT 3 instruction described on page 152. This one-byte INT 3 instruction differs from the two-byte INT 3 instruction in the following ways:

- The INT 3 interrupt is handled without any IOPL checking.
- In VME mode, the interrupt is not redirected via the interrupt redirection table.

| Mnemonic | Opcode | Description |
|----------|--------|----------------------------------|
| INT 3 | CC | Trap to debugger at Interrupt 3. |

For complete descriptions of the steps performed by INT instructions, see the following:

- *Legacy-Mode Interrupts*: “Legacy Protected-Mode Interrupt Control Transfers” in Volume 2.
- *Long-Mode Interrupts*: “Long-Mode Interrupt Control Transfers” in Volume 2.

Action

// See “Pseudocode Definitions” on page 46.

INT3_START:

```
IF (REAL_MODE)
    INT3_REAL
ELIF (PROTECTED_MODE)
    INT3_PROTECTED
ELSE // (VIRTUAL_MODE)
    INT3_VIRTUAL
```

INT3_REAL:

```
temp_RIP = READ_MEM.w [idt:3*4]
           // read target CS:RIP from the real-mode idt, entry 3
temp_CS  = READ_MEM.w [idt:3*4+2]

PUSH.w old_RFLAGS
PUSH.w old_CS
PUSH.w old_RIP

IF (temp_RIP>CS.limit)
    EXCEPTION [#GP]
```



```

CS.sel = temp_CS
CS.base = temp_CS SHL 4

RFLAGS.AC,TF,IF,RF cleared
RIP = temp_RIP
EXIT

```

INT3_PROTECTED:

```

temp_vector = 3    // go to int-3 handler
temp_idt_desc = READ_IDT (temp_vector)

IF (temp_idt_desc.attr.type = 'taskgate')
    TASK_SWITCH    // using tss selector in the task gate as the target tss

temp_RIP = temp_idt_desc.offset

IF (LONG_MODE)    // in long mode, we need to read the 2nd half of a 16-byte
interrupt-gate
{
    // from the gdt/ldt, to get the upper 32 bits of the
    // target RIP
    temp_upper = READ_MEM.q [idt:3*16+8]
    IF (temp_upper's extended attribute bits != 0)
        EXCEPTION [#GP(0)]    // the extended attribute bits must be all zero
    temp_RIP = temp_RIP + (temp_upper SHL 32) // concatenate both halves of RIP
}

IF (LONG_MODE)    // the size of the gate controls the size of the stack pushes
    V=8-byte    // long mode only uses 64-bit gates
ELIF ((temp_idt_desc.attr.type = 'intgate32')
    || (temp_idt_desc.attr.type = 'trapgate32'))
    V=4-byte    // legacy mode, using a 32-bit gate
ELSE // gate is intgate16 or trapgate16
    V=2-byte    // legacy mode, using a 16-bit gate

CS = READ_DESCRIPTOR (temp_idt_desc.segment, intcs_chk)

temp_CPL = CS.sel.rpl

IF (CPL=temp_CPL)
{
    IF (LONG_MODE)
    {
        IF (temp_idt_desc.ist!=0)
            // in long mode, if the idt gate specifies an ist pointer,
            // a stack-switch is always done
            RSP = READ_MEM.q [tss:ist_index*8+28]
            RSP = RSP AND 0xFFFFFFFFFFFFFFF0
            // in long mode, interrupts/exceptions align rsp to a
            // 16-byte boundary

```

```

        PUSH.q old_SS
            // in long mode, SS:RSP is always pushed to the stack
        PUSH.q old_RSP
    }

    PUSH.v old_RFLAGS
    PUSH.v old_CS
    PUSH.v old_RIP

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
    {
        EXCEPTION [#GP(0)]
    }

    RFLAGS.VM,NT,TF,RF cleared
    RFLAGS.IF cleared if interrupt gate

    RIP = temp_RIP
    EXIT
}
ELSE // (CPL > temp_CPL)
{
    CPL = temp_CPL

    temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER (CPL,
        temp_idt_desc.ist)

    IF (LONG_MODE)
        temp_RSP = temp_RSP AND 0xFFFFFFFFFFFFFFF0
            // in long mode, interrupts/exceptions round rsp to a
            // 16-byte boundary

    RSP.q = temp_RSP
    SS = temp_SS_desc

    PUSH.v old_SS
    PUSH.v old_RSP
    PUSH.v old_RFLAGS
    PUSH.v old_CS
    PUSH.v old_RIP

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
    {
        EXCEPTION [#GP(CS.sel)]
    }

    RFLAGS.VM,NT,TF,RF cleared
    RFLAGS.IF cleared if interrupt gate

```

```

    RIP = temp_RIP
    EXIT
}

```

INT3_VIRTUAL:

```

temp_vector = 3           // go to int-3 handler
temp_idt_desc = READ_IDT (temp_vector)

IF (temp_idt_desc.attr.type = 'taskgate')
    TASK_SWITCH           // using tss selector in the task gate as the
                        // target tss

IF ((temp_idt_desc.attr.type = 'intgate32')
    || (temp_idt_desc.attr.type = 'trapgate32'))
    // the size of the gate controls the size of the
    // stack pushes
{
    V=4-byte             // legacy mode, using a 32-bit gate
}
ELSE // gate is intgate16 or trapgate16
    V=2-byte             // legacy mode, using a 16-bit gate

temp_RIP = temp_idt_desc.offset
CS = READ_DESCRIPTOR (temp_idt_desc.sel, intcs_chk)

IF (CS.sel.rpl!=0)       // handler must run at cpl 0
    EXCEPTION [#GP(0)]

CPL = 0

temp_ist = 0             // call-far doesn't use ist pointers
temp_SS_desc:temp_RSP = READ_INNER_LEVEL_STACK_POINTER (CPL, temp_ist)

RSP.q = temp_RSP
SS = temp_SS_desc

PUSH.v old_GS
PUSH.v old_FS
PUSH.v old_DS
PUSH.v old_ES
PUSH.v old_SS
PUSH.v old_RSP
PUSH.v old_RFLAGS       // pushed with RF clear
PUSH.v old_CS
PUSH.v old_RIP

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(CS.sel)]

```

```

DS = NULL    // can't use virtual-mode selectors in protected mode
ES = NULL    // can't use virtual-mode selectors in protected mode
FS = NULL    // can't use virtual-mode selectors in protected mode
GS = NULL    // can't use virtual-mode selectors in protected mode

```

```

RFLAGS.VM,NT,TF,RF cleared
RFLAGS.IF cleared if interrupt gate

```

```

RIP = temp_RIP
EXIT

```

Related Instructions

INT, INTO, IRET

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|-----|-----|-----|-----|-------|----|----|-----|-----|----|----|----|----|----|
| | | | 1/0 | 1/0 | 1/0 | 1/0 | | | | 1/0 | 1/0 | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| Breakpoint, #BP | X | X | X | INT 3 instruction was executed. |
| Invalid TSS, #TS (selector) | | X | X | The TSS stack-segment selector was null. |
| | | X | X | The new stack-segment selector in the TSS contained an RPL that was not equal to the DPL of the code-segment descriptor accessed by the interrupt or trap gate. |
| | | X | X | The target stack-segment descriptor contained a DPL that was not equal to that of the code-segment descriptor accessed by the interrupt or trap gate. |
| | | X | X | The TSS stack-segment was not a writable data segment. |
| Segment not present, #NP (selector) | | X | X | The index in the stack-segment selector was beyond the descriptor table limits. |
| | | X | X | One of the following was not present: code segment, TSS, trap gate, task gate, or interrupt gate. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| Stack, #SS | X | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| Stack, #SS (selector) | | X | X | The stack segment limit was exceeded when a stack switch occurred. |
| | | | X | As part of a stack switch, the SS register was not present. |
| General protection, #GP | | X | X | The instruction pointer in the task gate, interrupt gate, trap gate, or IDT was outside the limits of the code segment. |
| | X | | | During instruction execution, the effective address of one of the operands pointed to an illegal memory location. |
| | X | | | The exception vector was beyond the limits of the IDT. |
| | | X | | The IOPL was less than 3 and the DPL of the interrupt-, trap-, or task-gate descriptor was not equal to 3. |
| General protection, #GP (selector) | | X | X | The segment selector was null in the trap, interrupt, or task gate. |
| | | X | X | The selector index for the TSS, gate, or code segment, was beyond the limits of the descriptor table. |
| | | X | X | The exception vector was beyond the limits of the IDT. |
| | | X | X | The descriptor in the IDT was not for an interrupt, trap, or task gate. |
| | | X | X | An interrupt was generated by the INTn instruction and the DPL of an interrupt, trap, or task gate was less than the CPL. (For protected mode, also INT 3 or INTO instruction) |
| | | X | X | The segment selector in a trap gate or interrupt did not point to a code-segment descriptor. |
| | | X | X | The TI bit in the TSS segment selector indicated a local table. |
| | | X | X | The segment descriptor for the TSS indicated that the TSS was not busy or not available. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |

INVD Invalidate Caches

The INVD instruction flushes internal caches (data cache, instruction cache, and on-chip L2 cache) and triggers a bus cycle that causes external caches to flush themselves as well. This instruction does not invalidate TLB caches.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| INVD | 0F 08 | Flush internal caches and trigger external cache flushes. |

No data is written back to main memory from flushed internal caches. After flushing internal caches, the processor proceeds immediately with the execution of the next instruction without waiting for external hardware to flush its caches.

This is a privileged instruction. The current privilege level (CPL) of a procedure invalidating the processor's internal caches must be zero.

To insure that data is written back to memory prior to invalidating caches, use the WBINVD instruction.

The INVD instruction is a serializing instruction.

Related Instructions

WBINVD

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--------------------|
| General protection, #GP | | X | X | CPL was not 0. |

INVLPG Invalidate TLB Entry

The INVLPG instruction takes a 1-byte operand and invalidates the TLB entry that would be used for that byte.

| Mnemonic | Opcode | Description |
|--------------------|----------|---|
| INVLPG <i>mem8</i> | 0F 01 /7 | Invalidate the TLB for the page containing a specified memory location. |

This instruction invalidates the TLB entry, regardless of the G (Global) bit setting in the associated PDE or PTE entry and regardless of the the page size (4K, 2M or 4M). It may invalidate any number of additional TLB entries, in addition to the targeted entry.

INVLPG is a serializing instruction and a privileged instruction. The current privilege level must be zero to execute this instruction in legacy protected mode or long mode.

See “Page Translation and Protection” in Volume 2 for more information on page translation.

Related Instructions

MOV CR n (CR3 and CR4), SWAPGS

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|------------------------------------|
| Invalid opcode, #UD | X | X | X | Destination operand is a register. |
| General protection, #GP | | X | X | CPL was not 0. |


```

    IRET_PROTECTED
ELSE // (VIRTUAL_MODE)
    IRET_VIRTUAL

```

```
IRET_REAL:
```

```

    POP.v temp_RIP
    POP.v temp_CS
    POP.v temp_RFLAGS

    IF (temp_RIP > CS.limit)
        EXCEPTION [#GP]

    CS.sel = temp_CS
    CS.base = temp_CS SHL 4

    RFLAGS.v = temp_RFLAGS // VIF,VIP,VM unchanged
    RIP = temp_RIP
    EXIT

```

```
IRET_PROTECTED:
```

```

    IF (RFLAGS.NT=1)                // iret does a task-switch to a previous task
        IF (LEGACY_MODE)
            TASK_SWITCH              // using the 'back link' field in the tss
        ELSE
            EXCEPTION [#GP(0)]      // (LONG_MODE)
            // task switches aren't supported in long mode

    POP.v temp_RIP
    POP.v temp_CS
    POP.v temp_RFLAGS

    IF ((temp_RFLAGS.VM=1) && (CPL=0) && (LEGACY_MODE))
        IRET_FROM_PROTECTED_TO_VIRTUAL

    temp_CPL = temp_CS.rpl

    IF ((64BIT_MODE) || (temp_CPL!=CPL))
    {
        POP.v temp_RSP              // in 64-bit mode, iret always pops ss:rsp
        POP.v temp_SS
    }

    CS = READ_DESCRIPTOR (temp_CS, iret_chk)

    IF ((64BIT_MODE) && (temp_RIP is non-canonical)
        || (!64BIT_MODE) && (temp_RIP > CS.limit))
    {
        EXCEPTION [#GP(0)]
    }

```

```

}

CPL = temp_CPL

IF ((started in 64-bit mode) || (changing cpl))
    // ss:rsp were popped, so load them into the registers
{
    SS = READ_DESCRIPTOR (temp_SS, ss_chk)
    RSP.s = temp_RSP
}

FOR (seg = ES, DS, FS, GS)
    IF ((seg.attr.dpl < CPL) && ((seg.attr.type = 'data')
        || (seg.attr.type = 'non-conforming-code')))
    {
        seg = NULL        // can't use lower dpl data segment at higher cpl
    }

RFLAGS.v = temp_RFLAGS    // VIF,VIP,IOPL only changed if (old_CPL=0)
                          // IF only changed if (old_CPL<=old_RFLAGS.IOPL)
                          // VM unchanged
                          // RF cleared

RIP = temp_RIP
EXIT

```

IRET_VIRTUAL:

```

IF ((RFLAGS.IOPL<3) && (CR4.VME=0))
    EXCEPTION [#GP(0)]

POP.v temp_RIP
POP.v temp_CS
POP.v temp_RFLAGS

IF (temp_RIP > CS.limit)
    EXCEPTION [#GP(0)]

IF (RFLAGS.IOPL=3)
{
    RFLAGS.v = temp_RFLAGS // VIF,VIP,VM,IOPL unchanged
                          // RF cleared

    CS.sel = temp_CS
    CS.base = temp_CS SHL 4

    RIP = temp_RIP
    EXIT
}

// now ((IOPL<3) && (CR4.VME=1))

```

```

    ELSIF ((OPERAND_SIZE=16) && !((temp_RFLAGS.IF=1)
        && (RFLAGS.VIP=1)) && (temp_RFLAGS.TF=0))
    {
        RFLAGS.w = temp_RFLAGS // RFLAGS.VIF=temp_rFLAGS.IF
                                // IF,VIP,VM,IOPL unchanged
                                // RF cleared

        CS.sel = temp_CS
        CS.base = temp_CS SHL 4

        RIP = temp_RIP
        EXIT
    }
    ELSE // ((RFLAGS.IOPL<3) && (CR4.VME=1) && ((OPERAND_SIZE=32) ||
((temp_RFLAGS.IF=1) && (RFLAGS.VIP=1)) || (temp_RFLAGS.TF=1)))
        EXCEPTION [#GP(0)]

```

IRET_FROM_PROTECTED_TO_VIRTUAL:

```

    // temp_RIP already popped
    // temp_CS already popped
    // temp_RFLAGS already popped, temp_RFLAGS.VM=1

    POP.d temp_RSP
    POP.d temp_SS
    POP.d temp_ES
    POP.d temp_DS
    POP.d temp_FS
    POP.d temp_GS

    CS.sel = temp_CS // force the segments to have virtual-mode values
    CS.base = temp_CS SHL 4
    CS.limit= 0x0000FFFF
    CS.attr = 16-bit dpl3 code

    SS.sel = temp_SS
    SS.base = temp_SS SHL 4
    SS.limit= 0x0000FFFF
    SS.attr = 16-bit dpl3 stack

    DS.sel = temp_DS
    DS.base = temp_DS SHL 4
    DS.limit= 0x0000FFFF
    DS.attr = 16-bit dpl3 data

    ES.sel = temp_ES
    ES.base = temp_ES SHL 4
    ES.limit= 0x0000FFFF
    ES.attr = 16-bit dpl3 data

    FS.sel = temp_FS

```

```

FS.base = temp_FS SHL 4
FS.limit= 0x0000FFFF
FS.attr = 16-bit dpl3 data

GS.sel  = temp_GS
GS.base = temp_GS SHL 4
GS.limit= 0x0000FFFF
GS.attr = 16-bit dpl3 data

RSP.d = temp_RSP
RFLAGS.d = temp_RFLAGS
CPL = 3

RIP = temp_RIP AND 0x0000FFFF
EXIT

```

Related Instructions

INT, INTO, INT3, RET, IRETD, IRETQ

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|-----|-----|-----|-----|-----|-----|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|--|
| Segment not present, #NP (selector) | | | X | The return code or stack segment was not present. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | | X | X | The return instruction pointer was not within the return code segment limit. |
| | | | X | The IOPL was not equal to 3. |
| | | | X | The return code or stack segment selector was null. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|--|------|-----------------|-----------|---|
| General protection, #GP (selector) | | | X | The segment selector index was outside the descriptor table limits. |
| | | | X | The return code segment selector RPL was greater than the CPL. |
| | | | X | The DPL for a nonconforming-code segment was not equal to the RPL of the code segment selector. |
| | | | X | The DPL for a conforming-code segment was greater than the return code segment selector RPL. |
| | | | X | The stack segment descriptor DPL was not equal to the RPL of the return code segment selector. |
| | | | X | The stack segment was not a writable data segment. |
| | | | X | The stack segment selector RPL was not equal to the RPL of the return code segment selector. |
| | | | X | The segment descriptor for a code segment selector did not indicate it was a code segment. |
| | | | X | The TI bit in the TSS's segment selector indicated a local table. |
| | | | X | The segment descriptor for the TSS indicated that the TSS was busy or not available. |
| General protection, segment overrun, #GP | X | X | | The return instruction pointer was not within the return code segment limit. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LAR Load Access Rights Byte

The LAR instruction loads the access rights from the segment descriptor specified by a 16-bit, 32-bit, or 64-bit bit source register or memory operand into a specified 16-bit general-purpose register and sets the zero (ZF) flag in the rFLAGS register if successful.

| Mnemonic | Opcode | Description |
|-----------------------------|----------|--|
| LAR <i>reg16, reg/mem16</i> | 0F 02 /r | Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with FF00h and saves the result in the 16-bit destination register. |
| LAR <i>reg32, reg/mem32</i> | 0F 02 /r | Reads the GDT/LDT descriptor referenced by the 32-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 32-bit destination register. |
| LAR <i>reg64, reg/mem64</i> | 0F 02 /r | Reads the GDT/LDT descriptor referenced by the 32-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 64-bit destination register. |

The LAR instruction performs the following:

- Checks that the segment selector is not null.
- Check whether the specified segment selector points to a descriptor that is within the limits of the global descriptor table or local descriptor table being accessed.
- Checks whether the specified segment descriptor is visible at the current privilege level (CPL). That is, for segments that are not conforming code segments, the RPL of both the CPL and selector must not be greater than the descriptor's DPL
- Checks that the descriptor type is valid for the LAR instruction. Valid descriptor types include the following:

| Valid Descriptor Type | Description |
|-----------------------|------------------------------|
| — | All code or data descriptors |
| 1 | Available 16-bit TSS |
| 2 | LDT |
| 3 | Busy 16-bit TSS |
| 4 | 16-bit call gate |
| 5 | 16-bit or 32-bit task gate |

| Valid Descriptor Type | Description |
|-----------------------|----------------------|
| 9 | Available 32-bit TSS |
| B | Busy 32-bit TSS |
| C | 32-bit call gate |

If the segment selector passes these checks, it is loaded into the destination general-purpose register. If it does not, then the zero flag is cleared and the destination register is not modified.

When the operand size is 16 bits, access rights include the DPL and Type fields located in bytes 4 and 5 of the descriptor table entry. Before loading the access rights into the destination operand, the low order word is masked with FF00H.

When the operand size is 32 or 64 bits, access rights include the DPL and type as well as the descriptor type (S field), segment present (P flag), available to system (AVL flag), default operation size (D/B flag), and granularity flags located in bytes 4–7 of the descriptor. Before being loaded into the destination operand, the doubleword is masked with 00FF_FF00H.

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

This instruction can only be executed in protected mode.

Related Instructions

ARPL, LSL, VERR, VERW

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | | | | | | 1/0 | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | | | X | During instruction execution, the effective address of one of the segment registers used for the operand pointed to an illegal memory location or a non-canonical address was generated. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LGDT**Load Global Descriptor Table Register**

Loads the pseudo-descriptor specified by the source operand into the global descriptor table register (GDTR). The pseudo-descriptor is a memory location containing the GDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is 6 bytes; in 64-bit mode, it is 10 bytes.

| Mnemonic | Opcode | Description |
|----------------------|----------|--|
| LGDT <i>mem16:32</i> | 0F 01 /2 | Loads <i>mem16:32</i> into the global descriptor table register. |
| LGDT <i>mem16:64</i> | 0F 01 /2 | Loads <i>mem16:64</i> into the global descriptor table register. |

If the operand size is 16 bits, the high-order byte of the 6-byte operand is not used. The lower 2 bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the GDTR is filled with zeros.

If the operand size is 32 bits, the lower 2 bytes specify the 16-bit limit and the upper 4 bytes specify a 32-bit base address.

In 64-bit mode, the lower 2 bytes specify the 16-bit limit and the upper 8 bytes specify a 64-bit base address. In 64-bit mode, size prefixes are ignored and therefore the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at privilege level 0. It must be executed at least once in real mode to initialize the processor before switching to protected mode.

This is a serializing instruction.

Related Instructions

LIDT, LLDT, LTR, SGDT, SIDT, STR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | | X | The source operand was not a memory location. |
| Stack, #SS | X | | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | X | X | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | | X | CPL was not 0. |
| | | | X | The LGDT base address was not canonical. |
| | | | X | The segment register used to access memory contained a null selector. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |

LIDT**Load Interrupt Descriptor Table Register**

Loads the pseudo-descriptor specified by the source operand into the interrupt descriptor table register (IDTR). The pseudo-descriptor is a memory location containing the IDTR base and limit. In legacy and compatibility mode, the pseudo-descriptor is six bytes; in 64-bit mode, it is 10 bytes.

| Mnemonic | Opcode | Description |
|----------------------|---------------|---|
| LIDT <i>mem16:32</i> | 0F 01 /3 | Loads <i>mem16:32</i> into the interrupt descriptor table register. |
| LIDT <i>mem16:64</i> | 0F 01 /3 | Loads <i>mem16:64</i> into the interrupt descriptor table register. |

If the operand size is 16 bits, the high-order byte of the 6-byte operand is not used. The lower 2 bytes specify the 16-bit limit and the third, fourth, and fifth bytes specify the 24-bit base address. The high-order byte of the IDTR is filled with zeros.

If the operand size is 32 bits, the lower 2 bytes specify the 16-bit limit and the upper 4 bytes specify a 32-bit base address. The high-order byte of the IDTR is filled with zeros.

In 64-bit mode, the lower 2 bytes specify the 16-bit limit, and the upper 8 bytes specify a 64-bit base address. In 64-bit mode, size prefixes are ignored and the operand size is forced to 64 bits. Therefore the pseudo-descriptor is always 10 bytes.

This instruction is only used in operating system software and must be executed at privilege level 0. It is normally executed at least once in real mode to initialize the processor before switching to protected mode.

This is a serializing instruction.

Related Instructions

LGDT, LLDT, LTR, SGDT, SIDT, STR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | | X | The source operand was not a memory location. |
| Stack, #SS | X | | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | X | X | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | X | X | CPL was not 0.. |
| | | | X | The segment register used to access memory contained a null selector. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |

LLDT**Load Local Descriptor Table Register**

Loads the descriptor pointed to by the specified segment selector into the local descriptor table register (LDTR). The processor uses the selector to locate the descriptor for the LDT in the global descriptor table. It then loads this descriptor into the LDTR.

| Mnemonic | Opcode | Description |
|-----------------------|----------|--|
| LLDT <i>reg/mem16</i> | 0F 00 /2 | Loads the segment select at a specified 16-bit memory location into the local descriptor table register. |

If the source operand is null, the LDTR is marked invalid and all references to descriptors in the LDT will generate a general protection exception (#GP), except for the LAR, VERR, VERW or LSL instructions.

In 64-bit mode, the instruction references a 16-byte descriptor that contains a 64-bit base. The LDT type (02H) is redefined in 64-bit mode for use as the 16-byte LDT descriptor.

This instruction must be executed in protected mode. It is only provided for use by operating system software at CPL 0.

This is a serializing instruction.

Related Instructions

LGDT, LIDT, LTR, SGDT, SIDT, SLDT, STR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Segment not present, #NP (selector) | | | X | The LDT descriptor was not present. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|------------------------------------|------|-----------------|-----------|---|
| General protection, #GP | | | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | | X | CPL was not 0. |
| | | | X | The segment register used to access memory contained a null selector. |
| General protection, #GP (selector) | | | X | The selector operand did not point into the global descriptor table or the entry in the GDT was not a local descriptor table. |
| | | | X | Segment selector was beyond GDT limit. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |

LMSW**Load Machine Status Word**

The LMSW instruction loads the contents of the 16-bit register or memory operand into bits 0 through 15 of the machine status word in register CR0. Only the protection enabled (PE), monitor coprocessor (MP), emulation (EM), and task switched (TS) bits of CR0 are modified. Additionally, LMSW can set CR0.PE, but cannot clear it.

| Mnemonic | Opcode | Description |
|-----------------------|----------|--|
| LMSW <i>reg/mem16</i> | 0F 01 /6 | Loads the lower 4 bits of the source into the lower 4 bits of CR0. |

This instruction cannot be used in legacy real mode. The LMSW instruction can only be used when the current privilege level is zero. It is only provided for compatibility with early processors and should only be used in operating system software.

Use the MOV instruction to load all 32 bits of CR0.

Related Instructions

MOV, SMSW

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|---|
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | | X | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | | X | CPL was not 0. |
| | | | X | The segment register used to access memory contained a null selector. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |

LSL Load Segment Limit

The LSL instruction loads a general-purpose register with the segment limit from the segment descriptor specified by the segment selector in a 16-bit or 32-bit memory or register operand.

| Mnemonic | Opcode | Description |
|-----------------------------|----------|---|
| LSL <i>reg16, reg/mem16</i> | 0F 03 /r | Loads a 16-bit general-purpose register with segment limit for a selector specified in a 16-bit memory or register operand. |
| LSL <i>reg32, reg/mem32</i> | 0F 03 /r | Loads a 32-bit general-purpose register with segment limit for a selector specified in a 32-bit memory or register operand. |
| LSL <i>reg64, reg/mem32</i> | 0F 03 /r | Loads a 64-bit general-purpose register with segment limit for a selector specified in a 32-bit memory or register operand. |

In 64-bit mode, for both 32-bit and 64-bit operand sizes, 32-bit register results are zero-extended to 64 bits.

The LSL instruction performs the following:

- Checks to make sure that the segment selector exists.
- Check whether the specified segment selector points to a descriptor that is within the limits of the global descriptor table or local descriptor table being accessed.
- Checks whether the specified segment descriptor is visible at the current privilege level (CPL)
- Checks that the descriptor type is valid for the LAR instruction. Valid descriptor types are the following:

| Type | Name |
|------|----------------------------|
| 1 | Available 16-bit TSS |
| 2 | LDT |
| 3 | Busy 16-bit TSS |
| 4 | 16-bit call gate |
| 5 | 16-bit or 32-bit task gate |

| Type | Name |
|------|----------------------|
| 9 | Available 32-bit TSS |
| B | Busy 32-bit TSS |
| C | 32-bit call gate |

If the segment selector passes these checks and the segment limit is loaded into the destination general-purpose register, the zero flag of the rFLAGS register is set to 1. If it does not, then the zero flag is cleared and the segment limit is not loaded.

The segment limit is a 20-bit value. When loading the target register, this value is converted into a 32-bit value by shifting the 20-bit value to the left 12 bits and filling the low order 12 bits with 1s.

If the granularity bit is 1, a byte limit is loaded; if the granularity is zero, a page limit is loaded.

When the operand size is 16 bits, the upper 16 bits of the 32-bit adjusted segment limit are truncated and the lower 16-bits are loaded into the target register.

ZF (bit 6 of rFLAGS) is set if the segment limit is loaded successfully, otherwise it is cleared.

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | 0 | 0 | | | | | | | 1/0 | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| General protection, #GP | | | X | During instruction execution, the effective address of a stack operand pointed to an illegal stack-memory location. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

LTR Load Task Register

Loads the task state segment (TSS) descriptor pointed to by the specified segment selector into the task register (TR). The processor uses the selector to locate the descriptor for the TSS in the global descriptor table. It then loads this descriptor into the TR. The TSS is marked as busy, but no task switch is made.

| Mnemonic | Opcode | Description |
|----------------------|----------|--|
| LTR <i>reg/mem16</i> | 0F 00 /3 | Load the contents of a 16-bit register or memory operand into the task register. |

If the source operand is null, a general protection exception (#GP) is generated.

In 64-bit mode, the instruction references a 64-bit mode descriptor to load a 64-bit base. The TSS type (09H) is redefined in 64-bit mode for use as the 16-byte TSS descriptor.

This instruction must be executed in protected mode when the current privilege level is zero. It is only provided for use by operating system software.

The operand size attribute has no effect on this instruction.

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Segment not present, #NP (selector) | | | X | The TSS was not present. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|------------------------------------|------|-----------------|-----------|--|
| General protection, #GP | X | X | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | X | X | CPL was not 0. |
| | | | X | The segment register used to access memory contained a null selector. |
| General protection, #GP (selector) | | | X | The source selector pointed to a segment that is not a TSS or to a TSS for a task that was busy. |
| | | | X | The selector pointed to the LDT or was beyond GDT limit. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |

MOV(CR_n) Move to/from Control Registers

Moves the contents of a 32-bit or 64-bit general-purpose register to a control register or vice versa.

In 64-bit mode, with the exception of CR8, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32 bits and the upper 32 bits is written with zeros for writes.

| Mnemonic | Opcode | Description |
|-----------------------------|----------|--|
| MOV CR _n , reg32 | 0F 22 /r | Move the contents of a 32-bit register to CR _n |
| MOV CR _n , reg64 | 0F 22 /r | Move the contents of a 64-bit register to CR _n |
| MOV reg32, CR _n | 0F 20 /r | Move the contents of CR _n to a 32-bit register. |
| MOV reg64, CR _n | 0F 20 /r | Move the contents of CR _n to a 64-bit register. |

Note:

CR0, CR2, CR3, CR4, and CR8 are the only registers to which this instruction applies. See text for details.

Reading or writing to CR8 requires a REX prefix, and thus can only be accessed in 64-bit mode. However, it can also be read and modified using the task priority register described in “System-Control Registers” in Volume 2.

CR0 maintains the state of various control bits. CR2 and CR3 are used for page translation. CR4 holds various feature enable bits. CR8 is used to prioritize external interrupts. CR1, CR5, CR6, CR7, and CR9 through CR15 are all reserved and raise an undefined opcode exception (#UD) if referenced.

This instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

This is a serializing instruction.

MOV(CR_n) is a privileged instruction and must always be executed at CPL = 0.

Related Instructions

MOV

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|---|
| General protection, #GP | | X | X | CPL was not 0. |
| | | | X | An attempt was made to write reserved bits in the page-directory pointers table used in the extended physical addressing mode when the PAE flag in CR4 and the PG flag in control register CR0 were set to 1. |
| | X | | X | An attempt was made to write 1 to any reserved bit in CR4. |
| | | | X | An attempt was made to set CR0.PG when CR0.PE = 0. |

MOV(DR n) Move to/from Debug Registers

Debug registers are 64-bit registers. This instruction moves the contents of a debug register into a 32-bit or 64-bit general-purpose register or vice versa.

| Mnemonic | Opcode | Description |
|----------------------------|----------|---|
| MOV <i>reg32</i> , DR0–DR7 | 0F 21 /r | Move the contents of a debug register to a 32-bit register. |
| MOV <i>reg64</i> , DR0–DR7 | 0F 21 /r | Move the contents of a debug register to a 64-bit register. |
| MOV DR0–DR7, <i>reg32</i> | 0F 23 /r | Move the contents of a 32-bit register to a debug register. |
| MOV DR0–DR7, <i>reg64</i> | 0F 23 /r | Move the contents of a 64-bit register to a debug register. |

In non-64-bit mode, the operand size is fixed at 32-bits. Writes set the upper halves of the debug register to 0; reads only return the lower 32 bits. In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix.

DR8 through DR15 are reserved and generate an undefined opcode exception if referenced.

This is a serializing instruction. MOV(DR) is also a privileged instruction and must be executed at CPL 0.

The MOV(DR) instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

See “Debug and Performance Resources” in Volume 2 for details.

Related Instructions

MOV

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Debug, #DB | X | | X | A debug register was accessed while the GD flag in debug register DR7 was set. |
| Invalid opcode, #UD | X | X | X | The debug extensions (DE) bit in CR4 was set and a MOV instruction was executed involving DR4 or DR5. One of DR8-DR15 was referenced. |
| General protection, #GP | | | X | A 1 was written to any of the upper 32 bits of DR6 or DR7 in 64-bit mode. |
| | | X | X | CPL was not 0. |

RDMSR**Read Model-Specific Register**

The RDMSR instruction loads the contents of a 64-bit model-specific register (MSR) specified in the ECX register into registers EDX:EAX. The EDX register receives the high-order 32 bits and the EAX register receives the low order bits. The RDMSR instruction is a 32-bit instruction, regardless of operand size; ECX always holds the MSR number, and EDX:EAX holds the data. In 64-bit mode, register results are zero-extended to 64 bits. If a model-specific register has fewer than 64 bits, the unimplemented bit positions loaded into the destination register are undefined.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| RDMSR | 0F 32 | Copy MSR specified by rCX into rDX:rAX. |

This instruction must be executed at a privilege level of 0 or a general protection fault #GP(0) will be raised. This exception is also generated if an attempt is made to specify a reserved or unimplemented model-specific register in ECX.

Use the CUID instruction to determine if this instruction is supported.

This is a serializing instruction.

For more information about model-specific registers, see the documentation for various hardware implementations and Volume 2, *System Programming*.

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|---|
| General protection, #GP | X | | X | The value in ECX specifies a reserved or unimplemented MSR address. |
| | | X | X | CPL was not 0. |

RDPMC**Read Performance-Monitoring Counter**

The RDPMC instruction loads the contents of the 64-bit PerfCtrn register specified by the ECX register (which contains the MSR number for the performance counter) into the EDX and EAX registers. The high-32 bits are loaded into EDX, and the low-32 bits are loaded into EAX. In 64-bit mode, 32-bit register results are zero-extended to 64 bits.

| Mnemonic | Opcode | Description |
|-----------------|---------------|---|
| RDPMC | 0F 33 | Copy the performance monitor counter specified by ECX into EDX:EAX. |

The RDPMC instruction always ignores operand size and always uses the 32-bit ECX, EDX, and EAX registers.

The AMD x86-64 Architecture supports four performance counters: PerfCtr0 through PerfCntr3.

Programs running at any privilege level can read performance monitor counters if the PCE flag in CR4 is set.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the performance counter is read.

For more information about performance-counter registers, see the documentation for various hardware implementations and “Performance Counters” in Volume 2.

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------------|-------------|-------------------------|------------------|--|
| General Protection Fault (GP(0)) | X | X | X | The value in ECX was not 0 or 1. CPL was not 0 and CR4.PCE = 0. |

RDTSC**Read Time-Stamp Counter**

The RDTSC instruction loads the value of the processor's time-stamp counter into registers EDX:EAX.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| RDTSC | 0F 31 | Copy the time-stamp monitor counter into EDX:EAX. |

The time-stamp counter is contained in a 64-bit model-specific register (MSR). The processor sets the counter to 0 upon reset and increments the counter every clock cycle. INIT does not modify the TSC.

The high-order 32 bits are loaded into EDX, and the low-order 32 bits are loaded into the EAX register. This instruction ignores operand size.

When the time-stamp disable flag (TSD) in CR4 is set, the RDTSC instruction can only be used at privilege level 0. If the TSD flag is zero, this instruction can be used at any privilege level.

This instruction is not serializing. Therefore, there is no guarantee that all instructions have completed at the time the time-stamp counter is read.

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| General protection, #GP | X | X | X | The CR4.TSD was 1 and the CPL was not 0. |

RSM**Resume from System Management Mode**

The RSM instruction resumes an operating system or application procedure previously interrupted by a system management interrupt (SMI). The processor state is restored from the information saved when the SMI was taken. If the processor detects invalid state information in the SMM save area, it goes into a shutdown state.

| Mnemonic | Opcode | Description |
|----------|--------|---|
| RSM | 0F AA | Resume operation of an interrupted program. |

Any of the following conditions will cause RSM to shutdown:

- An illegal combination of flag settings in CR0.
- A reserved bit in CR0, CR3, CR4, DR6, DR7, or the extended feature enable register (EFER) is set to 1.
- The following bit setting combination occurs: SMM.EFER.LME = 1, SMM.CR0.PG = 1, SMM.CR4.PAE = 0.

The x86-64 architecture uses a new 64-bit system management mode (SMM) state-save memory image. This 64-bit save-state map is used in all modes, regardless of mode. See “System-Management Mode” in Volume 2 for details.

rFLAGS Affected

None. Flags are restored from the state-save map.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | X | The processor is not in the System Management Mode (SMM). |

SGDT

Store Global Descriptor Table Register

Stores the global descriptor table register (GDTR) into the destination operand. In legacy and compatibility mode, the destination operand is six bytes; in 64-bit mode, it is 10 bytes. In all modes, size prefixes are ignored.

| Mnemonic | Opcode | Description |
|----------------------|----------|---|
| SGDT <i>mem16:32</i> | 0F 01 /0 | Store global descriptor table register to memory. |
| SGDT <i>mem16:64</i> | 0F 01 /0 | Store global descriptor table register to memory. |

In legacy or compatibility mode, the lower two bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the lower 32-bits of the base address.

In 64-bit mode, the lower 2 bytes specify the 16-bit limit and the upper 8 bytes specify a 64-bit base address.

Behavior of the SGDT instruction depends on whether the mode is 64-bit mode, but SGDT completely ignores operand size.

This instruction is intended for use in operating system software, but its use in application programs at any CPL is allowed.

Related Instructions

SIDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | X | The destination operand was a register. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | X | | X | During instruction execution, the data segment limit was exceeded or non-canonical address was encountered. |
| | | | X | The destination operand was in a nonwritable segment. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SIDT Store Interrupt Descriptor Table Register

Stores the interrupt descriptor table register (IDTR) in the destination operand. In legacy and compatibility mode, the destination operand is 6 bytes; in 64-bit mode it is 10 bytes. In all modes, size prefixes are ignored.

| Mnemonic | Opcode | Description |
|----------------------|----------|---|
| SIDT <i>mem16:32</i> | 0F 01 /0 | Store global descriptor table register to memory. |
| SIDT <i>mem16:64</i> | 0F 01 /0 | Store global descriptor table register to memory. |

In legacy or compatibility mode, the lower 2 bytes of the operand specify the 16-bit limit and the upper 4 bytes specify the lower 32-bits of the base address.

In 64-bit mode, the lower 2 bytes specify the 16-bit limit and the upper 8 bytes specify a 64-bit base address.

This instruction is intended for use in operating system software, but its use in application programs is allowed.

Related Instructions

SGDT, SLDT, STR, LGDT, LIDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The destination operand was a register. |
| Stack, #SS | X | X | X | A memory address exceeded the stack segment limit or was non-canonical . |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| General protection, #GP | X | X | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | | X | The destination operand was in a nonwritable segment. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SLDT**Store Local Descriptor Table Register**

The SLDT instruction stores the segment selector from the local descriptor table (LDT) selector to a register or memory destination operand.

| Mnemonic | Opcode | Description |
|-------------------|----------|---|
| SLDT <i>reg16</i> | 0F 00 /0 | Store local descriptor table register to a 16-bit register. |
| SLDT <i>reg32</i> | 0F 00 /0 | Store local descriptor table register to a 32-bit register. |
| SLDT <i>reg64</i> | 0F 00 /0 | Store local descriptor table register to a 64-bit register. |
| SLDT <i>mem16</i> | 0F 00 /0 | Store local descriptor table register to a 16-bit memory operand. |

If the destination operand is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

If the operand is a register, the (16-bit) selector is zero-extended into a 16-, 32-, or 64-bit general purpose register, depending on operand size. If the operand is a 32- or 64-bit register, the segment descriptor is written to the lower 16 bits and the upper bits are filled with zeros.

With a 64-bit operand, this instruction performs the same operation as in legacy mode, except that it zero-extends the 2-byte LDT selector to 64 bits.

The MOV (MOV DS, etc.) instruction can be used more efficiently to load a segment selector to a register or memory.

This instruction can be issued only in protected mode. It can be executed at any privilege level.

Related Instructions

SIDT, SGDT, STR, LIDT, LGDT, LLDT, LTR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical. |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a nonwritable segment. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SMSW**Store Machine Status Word**

Stores bits 0 through 15 of the machine status word (CR0). The target can be a 16-bit, 32-bit, or 64-bit register or memory operand or a 16-bit memory operand. With a 32-bit or 64-bit register operand, the low order 16-bits of CR0 are copied into the low order bits of the register with zero extension into the upper 16 or 48 bits.

| Mnemonic | Opcode | Description |
|-------------------|---------------|---|
| <i>SMSW reg16</i> | 0F 01 /4 | Stores the machine status word to a 16-bit register operand. |
| <i>SMSW reg32</i> | 0F 01 /4 | Stores the machine status word in the low-order 16 bits of a 32-bit register. |
| <i>SMSW reg64</i> | 0F 01 /4 | Stores the machine status word in the low-order 16 bits of a 64-bit register. |
| <i>SMSW mem16</i> | 0F 01 /4 | Stores the low order 16 bits of the machine status word in memory. |

In 64-bit mode, this instruction performs the same operation as in legacy mode, except that it zero-extends the 2-byte MSW to 64 bits.

This instruction is provided for compatibility with early processors and should only be used in operating system software. Current practice is to use the MOV(CR n) instruction to load all 32 bits of CR0.

This instruction can be used at any privilege level (CPL).

Related Instructions

LMSW, MOV(CR n)

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Stack, #SS | X | X | X | A memory address exceeds the stack segment limit or is non-canonical. |
| General protection, #GP | | | X | During instruction execution, the data segment limit was exceeded or a non-canonical address was encountered. |
| | | | X | The destination operand is in a nonwritable segment. |
| | | | X | There is a null segment selector in a data-segment register that is accessing memory. |
| Page fault, #PF | | X | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | X | X | An unaligned-memory data reference was performed while alignment checking is enabled. |

STI Set Interrupt Flag

Sets the interrupt flag (IF) to 1, allowing a maskable hardware interrupt to be recognized through interrupt request (INTR). INTR is an input signal into the processor, requesting suspension of execution of the current program. It is used for external hardware interrupts. The interrupt flag (IF) is used to mask or unmask the INTR signal. When the INTR is masked (IF = 0), the CPU does not acknowledge the interrupt. When the INTR is unmasked (IF = 1), the CPU will acknowledge the interrupt request and suspend the current process.

| Mnemonic | Opcode | Description |
|----------|--------|---------------------|
| STI | FB | Set interrupt flag. |

If the current privilege level (CPL) is less than IOPL, STI sets the IF flage to 1. EFLAGS.VIF is set, in virtual mode, if CR4.VME is set (1), or in protected mode,if CR4.PVI is set and CPL = 3. Otherwise, a GP# fault will be generated.

If STI sets IF and it was already clear, then interrupts are not enabled until after the instruction following STI . Thus, if IF is 0, the CLI in the following code sequence has no effect.

```
;; IF=0 before STI!
STI      ;Set IF=1.
CLI      ;Clear IF.
NOP      ;IF=1! CLI has no effect.
```

In the following sequence, CLI clears IF since it no longer immediately follows STI.

```
;; IF=0 before STI!
STI      ;Set IF=1.
NOP      ;IF=1! CLI has no effect.
CLI      ;Clear IF.
```

If STI sets the VIF flag and VIP is already set, a GP# fault will be generated before the next instruction.

Related Instructions

CLI, INT

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|-----|----|----|----|----|----|----|
| | | 1/0 | | | | | | | | 1/0 | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. 1/0 is set to either one or zero. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| General protection, #GP | | X | X | The rFLAGS VIF and VIP flags are set to 1. |

STR Store Task Register

The STR instruction stores the segment selector from the task register to a 16-, 32-, or 64-bit general-purpose register or 16-bit memory operand.

| Mnemonic | Opcode | Description |
|------------------|----------|--|
| STR <i>mem16</i> | 0F 00 /1 | Stores the segment selector from the task register to a 16-bit memory location. |
| STR <i>reg16</i> | 0F 00 /1 | Stores the segment selector from the task register to a 16-bit general-purpose register. |
| STR <i>reg32</i> | 0F 00 /1 | Stores the segment selector from the task register to a 32-bit general-purpose register. |
| STR <i>reg64</i> | 0F 00 /1 | Stores the segment selector from the task register to a 64-bit general-purpose register. |

If the destination is a memory location, the segment selector is written to memory as a 16-bit value, regardless of operand size.

If the operand is a 32-bit register, the segment selector is written to the lower 16 bits, and the upper bits are filled with zeros.

If the operand is a 64-bit register, this instruction performs the same operation as in legacy mode, except that it zero-extends the 2-byte TR selector to 64 bits.

The STR instruction can only be used in protected mode, and it can be used at any privilege level. It is useful only in operation system software.

Related Instructions

LTR, LGDT, LIDT, LLDT, LTR, SIDT, SGDT

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | The destination operand was in a nonwritable segment. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

SWAPGS**Swap GS Register with KernelGSbase MSR**

The SWAPGS instruction provides a fast method for system software to load a pointer to system data-structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction, or as a result of an interrupt or exception. Prior to returning to application software, SWAPGS can be used to restore the application data pointer that was replaced by the system data-structure pointer.

| Mnemonic | Opcode | Description |
|----------|----------|--|
| SWAPGS | 0F 01 F8 | Save GS selector register with KernelGSbase MSR. (Invalid in legacy and compatibility modes.) |

The SWAPGS instruction only exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden-portion of the GS selector register (GS.base). This allows the system-kernel software to access kernel data-structures by using the GS segment-override prefix during memory references.

The address stored in the KernelGSbase MSR must be in canonical form. The WRMSR instruction used to load the KernelGSbase MSR causes a general-protection exception if the address loaded is not in canonical form. The SWAPGS instruction itself does not perform a canonical check.

This instruction is only valid in 64-bit mode at CPL = 0. A general protection exception is generated if this instruction is executed at any other privilege level.

For additional information about this instruction, refer to “System-Management Instructions” in Volume 2.

The opcode for SWAPGS is located in the INVLPG reg space.

| Opcode | MOD R/M byte | | | Instruction | |
|--------|--------------|-----|--------|-----------------|-------------|
| | MOD | REG | R/M | non 64-bit mode | 64-bit mode |
| 0F 01 | MOD != 11 | 111 | xxx | INVLPG | INVLPG |
| | 11 | 111 | 000 | #UD | SWAPGS |
| | 11 | 111 | != 000 | #UD | #UD |

Examples

1. At a kernel entry point, the OS uses SwapGS to obtain a pointer to kernel data structures and simultaneously save the user's GS base. Upon exit, it uses SwapGS to restore the user's GS base:

```

SystemCallEntryPoint:
SwapGS                ; get kernel pointer, save user GSbase
mov gs:[SavedUserRSP], rsp ; save user's stack pointer
mov rsp, gs:[KernelStackPtr] ; set up kernel stack
push rax              ; now save user GPRs on kernel stack
                    . ; perform system service
                    .
SwapGS                ; restore user GS, save kernel pointer

```

Related Instructions

None

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | X | X | X | Software was executing in legacy or compatibility mode. |
| General protection, #GP | | | X | CPL was not 0. |

SYSCALL **Fast System Call**

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions, which assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in any operating mode, but they are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

| Mnemonic | Opcode | Description |
|-----------------|---------------|------------------------|
| SYSCALL | OF 05 | Call operating system. |

The SYSCALL instruction transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes considerably fewer clock cycles than the CALL FAR /RET FAR instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of three to a new privilege level of zero. It is also assumed (but not checked) that the privilege level of the selectors for the CALL and RET target CS segments are set to zero and three, respectively (for SYSCALL the code segment selector requested privilege level is zero, and for SYSRET the code segment selector requested privilege level is three).

The SYSCALL and SYSRET instructions should be used in environments where neither the operating system nor applications alter the Code Segment or Stack Segment in a manner different than that assumed by the instructions. The SYSCALL instruction sets the base, limit, and attributes of both CS and SS to fixed values on entry. The SYSRET instruction sets CS to the same fixed values upon exiting. No memory accesses to descriptors are required. It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

Legacy x86 Mode. In legacy x86 mode, when SYSCALL is executed, the EIP register is copied into the ECX register. Bits 31–0 of the 64-bit SYSCALL/SYSRET target address register (STAR) are copied into the EIP register. (The STAR register is model-specific register C000_0081h.)

The IF and VM flags are cleared to 0 to disable interrupts and force the processor out of virtual-8086 mode.

New selectors are loaded, without permission checking (see above), as follows:

- Bits 47–32 of the STAR register specify the selector that is copied into the CS register.
- Bits 47–32 + 1000b of the STAR register specify the selector that is copied into the SS register.
- The CS_base and the SS_base are both cleared to zero.
- The CS_limit and the SS_limit are both set to 4 Gbyte.
- The CS segment attributes are set to execute/read with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by a 32-bit stack selector.

The CPL is set to 0, regardless of the value of bits 33–32 of the STAR register. There are no permission checks based on the CPL, real mode, or virtual-8086 mode.

The operating system must ensure that the descriptors corresponding to the selectors in the STAR register are consistent with the corresponding on-chip values (such as, base, limit, and attributes) that are set by the SYSCALL and SYSRET instructions.

Long Mode. When long mode is activated, the behavior of the SYSCALL instruction depends on whether the calling software is in 64-bit mode or compatibility mode.

In 64-bit mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from LSTAR bits 63–0.

In compatibility mode, SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from CSTAR bits 63–0.

- The CS_base and the SS_base are both cleared to zero.
- The CS_limit and the SS_limit are both set to 4 Gbyte.
- The CS segment attributes are set to execute/read for 64-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 64-bit stack referenced by RSP.

The WRMSR instruction is used to load the target RIP into the LSTAR or CSTAR registers. If the RIP written by WRMSR is not in canonical form, a general-protection exception (#GP) occurs. The SYSCALL and SYSRET CS and SS selectors used when long mode is active are stored in the STAR.

In long mode, the SYSCALL_FLAG_MASK register is used to specify which rFLAGS bits are cleared during a SYSCALL. How SYSCALL and SYSRET handle rFLAGS, depends on the processor's operating mode:

How SYSCALL handles EFLAGS, depends on the processor's operating mode:

In legacy mode, SYSCALL does the following with EFLAGS:

- EFLAGS.IF is cleared to 0.
- EFLAGS.RF is cleared to 0.

In long mode, SYSCALL does the following with EFLAGS:

- The current value of RFLAGS is saved in R11.
- RFLAGS.RF is cleared to 0.
- RFLAGS is masked using the value stored in SYSCALL_FLAG_MASK.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, and CSTAR), see “Fast System Call and Return” in Volume 2.

Action

// See “Pseudocode Definitions” on page 46.

SYSCALL_START:

```

    IF (MSR_EFER.SCE = 0)           // check if syscall/sysret are enabled
        EXCEPTION [#UD]

    IF (LONG_MODE)
        SYSCALL_LONG_MODE
    ELSE // (LEGACY_MODE)
        SYSCALL_LEGACY_MODE

```

SYSCALL_LONG_MODE:

```

    RCX.q = next_RIP
    R11.q = RFLAGS    // with rf cleared

    IF (64BIT_MODE)
        temp_RIP.q = MSR_LSTAR
    ELSE // (COMPATIBILITY_MODE)
        temp_RIP.q = MSR_CSTAR

```

```
CS.sel    = MSR_STAR.SYSCALL_CS AND 0xFFFC
CS.attr   = 64-bit code,dp10 // always switch to 64-bit mode in long mode
CS.base   = 0x00000000
CS.limit  = 0xFFFFFFFF
```

```
SS.sel    = MSR_STAR.SYSCALL_CS + 8
SS.attr   = 64-bit stack,dp10
SS.base   = 0x00000000
SS.limit  = 0xFFFFFFFF
```

```
RFLAGS = RFLAGS AND ~MSR_SFMASK
RFLAGS.RF = 0
```

```
CPL = 0
```

```
RIP = temp_RIP
EXIT
```

SYSCALL_LEGACY_MODE:

```
RCX.d = next_RIP
```

```
temp_RIP.d = MSR_STAR.eip
```

```
CS.sel    = MSR_STAR.syscall_cs AND 0xFFFC
CS.attr   = 32-bit code,dp10 // always switch to 32-bit mode in legacy mode
CS.base   = 0x00000000
CS.limit  = 0xFFFFFFFF
```

```
SS.sel    = MSR_STAR.syscall_cs + 8
SS.attr   = 32-bit stack,dp10
SS.base   = 0x00000000
SS.limit  = 0xFFFFFFFF
```

```
RFLAGS.vm,if,rf=0
```

```
CPL = 0
```

```
RIP = temp_RIP
EXIT
```

Related Instructions

SYSRET, SYSENTER, SYSEXIT

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|-----|-----|-----|-----|-----|-----|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The SYSCALL and SYSRET instructions are not supported, as indicated by bit 11 of CPUID extended function 8000_0001. |
| | X | X | X | The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.) |
| General Protection, #GP | | | X | The RIP written by WRMSR was not in canonical form. |

SYSENTER System Call

The SYSENTER instruction transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model.

| Mnemonic | Opcode | Description |
|----------|--------|------------------------|
| SYSENTER | 0F 34 | Call operating system. |

Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction, as well as the CS and SS selectors of the called and returned procedures:

- SYSENTER Target CS: Contains the CS selector of the called procedure. The SS selector is set to `sysenter.cs + 8`.
- SYSENTER Target ESP: Contains the called procedure's stack pointer.
- SYSENTER Target EIP: Contains the offset into the CS of the called procedure.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 CALL instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are cleared to 0.
- The CS and SS limit values are set to 4 Gbytes.
- The CS segment attributes are set to execute/read 32-bit code with a CPL of zero.
- The SS segment attributes are set to read/write and expand-up with a 32-bit stack referenced by a 32-bit stack selector.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The return EIP and application stack are not saved by this instruction. System software must explicitly save that information.

An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSCALL (and SYSRET) instructions when running in long mode.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Related Instructions

SYSCALL, SYSEXIT, SYSRET

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | 0 | | | | | | 0 | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|--------|-----------------|-----------|---|
| Invalid opcode, #UD | | | X X | Software was executing in long mode. The SYSENTER and SYSEXIT instructions are not supported, as indicated by bit 11 of CPUID standard function 1. |
| General protection, #GP | X X | | X | Protected mode was not enabled. The SYSENTER_CS MSR was cleared to 0. |

SYSEXIT System Return

The SYSEXIT instruction returns from the operating system to an application. It is a low-latency system return instruction designed for use by system and application software implementing a flat-segment memory model. This is a privileged instruction. The current privilege level must be zero to execute this instruction. An invalid-opcode exception occurs if this instruction is used in long mode. Software should use the SYSRET (and SYSCALL) instructions when running in long mode.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| SYSEXIT | OF 35 | Return from operating system to application. |

When a system procedure performs a SYSEXIT back to application software, the CS selector is updated to point to the second descriptor entry after the SYSENTER CS value (SYSENTER_CS+16). The SS selector is updated to point to the third descriptor entry after the SYSENTER CS value (SYSENTER_CS+24). The CPL is forced to 3, as are the descriptor privilege levels.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS and SS base values are cleared to 0.
- The CS and SS limit values are set to 4 Gbytes.
- The CS segment attributes are set to 32-bit read/execute, non-conforming at CPL 3.
- The SS segment attributes are set for a 32-bit stack segment, read/write and expand-up.

System software must create corresponding descriptor-table entries referenced by the new CS and SS selectors that match the values described above.

The following additional actions result from executing SYSEXIT:

- EIP is loaded from EDX.
- ESP is loaded from ECX.

System software must explicitly load the return address and application software-stack pointer into the EDX and ECX registers prior to executing SYSEXIT.

For additional information on this instruction, see “SYSENTER and SYSEXIT (Legacy Mode Only)” in Volume 2.

Related Instructions

SYSCALL, SYSENTER, SYSRET

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|----|----|----|----|
| | | | | | 0 | | | | | | | | | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| Invalid opcode, #UD | | | X X | Software was executing in long mode. The SYSENTER and SYSEXIT instructions are not supported, as indicated by bit 11 of CPUID standard function 1. |
| General protection, #GP | X | X | X X | Protected mode was not enabled. CPL was not 0.. The SYSENTER_S MSR was cleared to 0. |

SYSRET

Fast System Return

The SYSCALL and SYSRET instructions are low-latency system call and return control-transfer instructions, which assume that the operating system implements a flat-segment memory model. By eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions), calls to and returns from the operating system are greatly simplified. These instructions can be used in any operating mode, but they are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-segment memory model.

| Mnemonic | Opcode | Description |
|----------|--------|-------------------------------|
| SYSRET | OF 07 | Return from operating system. |

The SYSCALL instruction transfers control to a fixed entry point in an operating system. It is designed for use by system and application software implementing a flat-segment memory model.

This instruction has been optimized by reducing the number of checks and memory references that are normally made so that a call or return takes substantially fewer internal clock cycles when compared to the CALL/RET instruction method.

It is assumed that the base, limit, and attributes of the Code Segment will remain flat for all processes and for the operating system, and that only the current privilege level for the selector of the calling process should be changed from a current privilege level of 0 to a new privilege level of 3. It is also assumed (but not checked) that the privilege level of the selectors for the CALL and RET target CS segments are set to zero and three, respectively (for SYSCALL the code segment selector requested privilege level is 0, and for SYSRET the code segment selector requested privilege level is three).

The SYSCALL and SYSRET instructions should be used in environments where neither the operating system nor applications alter the Code Segment or Stack Segment in a manner different than that assumed by the instructions. The SYSCALL instruction sets the base, limit, and attributes of both CS and SS to fixed values on entry. The SYSRET instruction sets CS to the same fixed values upon exiting. In a flat memory model, only the privilege level of the segments needs to change. No memory accesses to descriptors are required. It is the responsibility of the operating system to keep the descriptors in memory that correspond to the CS and SS selectors loaded by the SYSCALL and SYSRET instructions consistent with the segment base, limit, and attribute values forced by these instructions.

When a system procedure performs a SYSRET back to application software, the CS selector is updated from bits 63–50 of the STAR register (SYSRET.CS) as follows:

- If the return is to 32-bit mode (legacy or compatibility), CS is updated with the value of SYSRET.CS.
- If the return is to 64-bit mode, CS is updated with the value of SYSRET.CS + 10h.

In both cases, the CPL is forced to 3, effectively ignoring STAR bits 49–48. The SS selector is updated to point to the next descriptor-table entry after the CS descriptor (SYSRET.CS + 8h), and its CPL is also forced to 3.

The hidden portions of the CS and SS segment registers are not loaded from the descriptor table as they would be using a legacy x86 RET instruction. Instead, the hidden portions are forced by the processor to the following values:

- The CS value is cleared to 0.
- The CS limit value is set to 4 Gbytes.
- The CS segment attributes are set to read-only at CPL = 3.
- The SS segment attributes are set to read/write and expand-up.

System software must create corresponding descriptor-table entries referenced by the STAR's SYSRET CS field that match the values described above.

When SYSCALLed system software is running in 64-bit mode, it has been entered from either 64-bit mode or compatibility mode. The corresponding SYSRET needs to know the mode to which it must return. Executing SYSRET in non-64-bit mode or with a 16- or 32-bit operand size, returns to 32-bit mode. Executing SYSRET in 64-bit mode with a 64-bit operand size returns to 64-bit mode.

The instruction pointer is updated with the return address based on the operating mode in which SYSRET is executed:

- If returning to 64-bit mode, SYSRET loads RIP with the value of RCX.
- If returning to 32-bit mode, SYSRET loads EIP with the value of ECX.

How SYSRET handles EFLAGS, depends on the processor's operating mode:

- In long mode, SYSRET loads the lower-32 RFLAGS bits from R11[31:0] and clears the upper 32 RFLAGS bits.
- In legacy mode, SYSRET sets EFLAGS.IF.

For further details on the SYSCALL and SYSRET instructions and their associated MSR registers (STAR, LSTAR, and CSTAR), see “Fast System Call and Return” in Volume 2.

Action

// See “Pseudocode Definitions” on page 46.

SYSRET_START:

```

    IF (MSR_EFER.SCE = 0)                // check if syscall/sysret are enabled
        EXCEPTION [#UD]

    IF (64BIT_MODE)
        SYSRET_64BIT_MODE
    ELSE // (!64BIT_MODE)
        SYSRET_NON_64BIT_MODE

```

SYSRET_64BIT_MODE:

```

    IF (CPL != 0)                        // sysret requires protected mode, cp10
        EXCEPTION [#GP(0)]

    IF (OPERAND_SIZE = 64)                // return to 64-bit mode
    {
        CS.sel  = (MSR_STAR.sysret_cs + 16) OR 3
        CS.base = 0x00000000
        CS.limit = 0xFFFFFFFF
        CS.attr  = 64-bit code,dp13

        temp_RIP.q = RCX
    }
    ELSE                                // return to 32-bit compatibility mode
    {
        CS.sel  = MSR_STAR.sysret_cs OR 3
        CS.base = 0x00000000
        CS.limit = 0xFFFFFFFF
        CS.attr  = 32-bit code,dp13

        temp_RIP.d = RCX
    }

    SS.sel  = MSR_STAR.sysret_cs + 8 // ss selector is changed,
                                    // ss base, limit, attributes unchanged

    RFLAGS.q = R11 // rf=0,vm=0
    CPL = 3

    RIP = temp_RIP
    EXIT

```

SYSRET_NON_64BIT_MODE:

```

    IF ((!PROTECTED_MODE) || (CPL != 0))
        // sysret requires protected mode, cp10

```

```

EXCEPTION [#GP(0)]

CS.sel    = MSR_STAR.sysret_cs OR 3
                                         // return to 32-bit legacy protected mode
CS.base   = 0x00000000
CS.limit  = 0xFFFFFFFF
CS.attr   = 32-bit code,dp13

temp_RIP.d = RCX

SS.sel    = MSR_STAR.sysret_cs + 8
                                         // ss selector is changed.
                                         // ss base, limit, attributes unchanged

RFLAGS.IF = 1
CPL = 3

RIP = temp_RIP
EXIT

```

Related Instructions

SYSCALL, SYSENTER, SYSEXIT

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|-----|-----|-----|-----|-----|-----|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Note:
 Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|--------------|-----------|--|
| Invalid opcode, #UD | X | X | X | The SYSENTER and SYSEXIT instructions are not supported, as indicated by bit 11 of CPUID standard function 1. |
| | X | X | X | The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.) |
| General protection, #GP | | X | X | CPL was not 0. |

UD2 Undefined Operation

Generates an invalid opcode exception. Unlike other undefined opcodes that may be defined as legal instructions in the future, UD2 is guaranteed to stay undefined.

| Mnemonic | Opcode | Description |
|----------|--------|------------------------------------|
| UD2 | 0F 0B | Raise an invalid opcode exception. |

Related Instructions

NOP

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|-----------------|-----------|-------------------------------------|
| Invalid opcode, #UD | X | X | X | This instruction is not recognized. |

VERR**Verify Segment for Reads**

Verifies whether a code or data segment specified by the segment selector in the 16-bit register or memory operand is readable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is readable. Otherwise, ZF is cleared. If the selector references a system descriptor, ZF is cleared, because system descriptors are never readable.

MnemonicVERR *reg/mem16***Opcode**

0F 00 /4

Description

Set the zero flag (ZF) to 1 if the segment selected can be read.

The processor does not recognize the VERR instruction in real or virtual-8086 mode.

Related Instructions

VERW

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|----|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | | | | | | 1/0 | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

Notes: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is 1/0. Unaffected flags are blank. Undefined flags are U.

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|--------------|-----------|---|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or is non-canonical . |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

VERW**Verify Segment for Writes**

Verifies whether a code or data segment specified by the segment selector in the 16-bit register or memory operand is writable from the current privilege level. The zero flag (ZF) is set to 1 if the specified segment is writable. Otherwise, ZF is cleared. Since VERW will always clear ZF when checking system descriptors, which are never writable, code segments can never be verified as writable.

| Mnemonic | Opcode | Description |
|-----------------------|----------|---|
| VERW <i>reg/mem16</i> | 0F 00 /5 | Set the zero flag (ZF) to 1 if the segment selected can be written. |

The processor does not recognize the VERR instruction in real or virtual-8086 mode.

Related Instructions

VERR

rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|--|-----|-----|----|----|----|----|-------|----|----|----|----|----|-----|----|----|----|
| | | | | | | | | | | | | | 1/0 | | | |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |
| Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to one or zero is 1/0. Unaffected flags are blank. Undefined flags are U. | | | | | | | | | | | | | | | | |

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---------------------|------|--------------|-----------|--|
| Invalid opcode, #UD | X | X | | This instruction is only recognized in protected mode. |
| Stack, #SS | | | X | A memory address exceeded the stack segment limit or was non-canonical . |

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|-------------------------|------|-----------------|-----------|---|
| General protection, #GP | | | X | A memory address exceeded a data segment limit or was non-canonical. |
| | | | X | There was a null segment selector in a data-segment register that was accessing memory. |
| Page fault, #PF | | | X | A page fault resulted from the execution of the instruction. |
| Alignment check, #AC | | | X | An unaligned-memory data reference was performed while alignment checking was enabled. |

WBINVD**Writeback and Invalidate Caches**

The WBINVD instruction writes all modified cache lines in the internal caches back to main memory and invalidates (flushes) internal caches. It then triggers a bus cycle that causes external caches to invalidate themselves as well. After invalidating internal caches, the processor proceeds immediately with the execution of the next instruction without waiting for external hardware to invalidate its caches.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| WBINVD | 0F 09 | Write modified cache lines to main memory, invalidate internal caches, and trigger external cache flushes. |

The INVD instruction can be used when cache coherence with memory is not important.

This instruction never invalidates TLBs.

This instruction is a serializing instruction.

This is a privileged instruction. The current privilege level of a procedure invalidating the processor's internal caches must be zero.

Related Instructions

INVD

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|--------------------|
| General protection, #GP | | X | X | CPL was not 0. |

WRMSR

Write to Model-Specific Register

The WRMSR instruction can be used to write data to a 64-bit model-specific register (MSR). These registers are widely used in performance-monitoring and debugging applications, as well as testability and program execution tracing.

| Mnemonic | Opcode | Description |
|----------|--------|--|
| WRMSR | 0F 30 | Write EDX:EAX to the MSR specified by ECX. |

This instruction writes the contents of the EDX:EAX register pair into a 64-bit model-specific register specified in the ECX register. The 32 bits in the EDX register are mapped into the high-order bits of the model-specific register and the 32 bits in EAX form the low-order 32 bits.

This instruction must be executed at a privilege level of 0 or a general protection fault #GP(0) will be raised. This exception is also generated if an attempt is made to specify a reserved or unimplemented model-specific register in ECX.

This is a serializing instruction.

The CPUID instruction can provide model information useful in determining the existence of a particular MSR.

See Volume 2, *System Programming*, for more information about model-specific registers, machine check architecture, performance monitoring and debug registers.

Related Instructions

RDMSR

rFLAGS Affected

None

Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|----------------------------|------|-----------------|-----------|---|
| General protection, #GP | X | | X | The value in ECX specifies a reserved or unimplemented MSR address. |
| | | X | X | CPL was not 0. |
| | X | | X | Writing 1 to any bit that must be zero (MBZ) in the MSR. |

Appendix A Opcode and Operand Encodings

This section specifies the hexadecimal and/or binary encodings for the opcodes and the implicit operand references used in the x86-64 instruction set. For an overview of the instruction formats to which these encodings apply, see Chapter 1, “Instruction Formats.”

A.1 Opcode-Syntax Notation

The following notation is used in this section to specify opcodes and their operands:

- A* Direct address of operand is encoded in instruction without a ModRM byte. Complex addressing using the SIB byte cannot be done.
- C* Control register specified by the ModRM *reg* field.
- D* Debug register specified by the ModRM *reg* field.
- E* The opcode is followed by a ModRM byte that specifies a GPR or memory operand. Memory addresses can be computed from a segment register, SIB byte, and/or displacement.
- F* rFLAGS register.
- G* The ModRM *reg* field selects a GPR.
- I* Immediate value.
- J* The instruction includes a relative offset that is added to the rIP.
- M* A memory reference encoded in the ModRM byte.
- O* The offset of an operand is encoded in the instruction. There is no ModRM byte in the instruction. Complex addressing using the SIB byte cannot be done.
- P* The ModRM *reg* field specifies a 64-bit MMX register.
- Q* A ModRM byte specifies an MMX-register or memory operand. Memory addresses can be computed from a segment register, SIB byte, and/or displacement.

- R* The ModRM *mod* field must be 11b (i.e., only the register form of this field is valid).
- S* The ModRM *reg* field specifies a segment register.
- T* The ModRM *reg* field specifies a test register.
- V* The ModRM *reg* field specifies a 128-bit XMM register.
- W* A ModRM byte specifies an XMM or memory operand. Memory addresses can be computed from a segment register, SIB byte, and/or displacement.
- X* A memory operand is addressed by the DS.rSI registers. Used in string instructions.
- Y* A memory operand is addressed by the ES.rDI registers. Used in string instructions.
- a* Two 16-bit or 32-bit memory operands, depending on the effective operand size (used in BOUND).
- b* A byte, irrespective of the effective operand size.
- c* A byte or word, depending on the effective operand size.
- d* A doubleword (32 bits), irrespective of the effective operand size.
- dq* A double-quadword (128 bits), irrespective of the effective operand size.
- p* A 32-bit or 48-bit pointer, depending on the effective operand size.
- pd* A 128-bit double-precision floating-point vector operand (packed double).
- pi* A 64-bit MMX operand (packed integer).
- ps* A 128-bit single-precision floating-point vector operand (packed single).
- q* A quadword, irrespective of the effective operand size.
- s* A 6-byte pseudo-descriptor.
- sd* A scalar double-precision floating-point operand (scalar double).

- si* A scalar doubleword (32-bit) integer operand (scalar integer).
- ss* A scalar single-precision floating-point operand (scalar single).
- v* A word, doubleword, or quadword, depending on the effective operand size.
- w* A word, irrespective of the effective operand size.
- z* A word if the effective operand size is 16 bits, or a doubleword if the effective operand size is 32 or 64 bits.
- /n* A ModRM-byte *reg* field or SIB-byte *base* field that contains a value (*n*) between zero (binary 000) and 7 (binary 111).

For definitions of the mnemonics used to name registers, see “Summary of Registers and Data Types” on page 28.

A.2 Opcode Encodings

A.2.1 One-Byte Opcodes

Table 4-1 shows the one-byte opcodes in which the low nibble is in the range 0–7h. Table 4-2 on page 383 shows those opcodes in which the low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the specified range of the low nibble.

Table 4-1. One-Byte Opcodes, Low Nibble 0–7h

| Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------------------|---|---|---|---|---|---|-------------------------|------------------------|
| 0 | ADD Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, Ib rAX, Iz | | | | | | PUSH ES ³ | POP ES ³ |
| 1 | ADC Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, Ib rAX, Iz | | | | | | PUSH SS ³ | POP SS ³ |
| 2 | AND Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, Ib rAX, Iz | | | | | | seg ES ³ | DAA ³ |

Note:

1. Rows in this table show the high opcode nibble, columns show the low opcode nibble.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.
4. Valid only in 64-bit mode.
5. Used as REX prefixes in 64-bit mode.

Table 4-1. One-Byte Opcodes, Low Nibble 0–7h (continued)

| Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------------------|--|---|------------------------------|--|----------------------------|----------------------------|---|----------------------------|
| 3 | XOR Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, Ib rAX, Iz | | | | | | seg SS ³ | AAA ³ |
| 4 | INC ⁵ rAX rCX rDX rBX rSP rBP rSI rDI | | | | | | | |
| 5 | PUSH rAX/r8 rCX/r9 rDX/r10 rBX/r11 rSP/r12 rBP/r13 rSI/r14 rDI/r15 | | | | | | | |
| 6 | PUSHA/D ³ | POPA/D ³ | BOUND ³ Gv, Ma | ARPL ³ Ew, Rw MOVXSD ⁴ | seg FS | seg GS | operand size | address size |
| 7 | JO Jb | JNO Jb | JB Jb | JNB Jb | JZ Jb | JNZ Jb | JBE Jb | JNBE Jb |
| 8 | Group 1 ² Eb, Ib Ev, Iz Eb, Ib ³ Ev, Ib | | | | TEST Eb, Gb Ev, Gv | | XCHG Eb, Gb Ev, Gv | |
| 9 | NOP rAX/r8, rAX | XCHG rCX/r9, rAX rDX/r10, rAX rBX/r11, rAX rSP/r12, rAX rBP/r13, rAX rSI/r14, rAX rDI/r15, rAX | | | | | | |
| A | MOV AL, Ob rAX, Ov Ob, AL Ov, rAX | | | | MOVSB Xb, Yb | MOVSW/D/Q Xv, Yv | CMPSB Xb, Yb | CMPSW/D/Q Xv, Yv |
| B | MOV AL, Ib CL, Ib DL, Ib BL, Ib AH, Ib CH, Ib DH, Ib BH, Ib | | | | | | | |
| C | Group 2 ² Eb, Ib Ev, Ib | | RET near Iw | | LES ³ Gv, Mp | LDS ³ Gv, Mp | Group 11 ² Eb, Ib Ev, Iz | |
| D | Group 2 ² Eb, 1 Ev, 1 Eb, CL Ev, CL | | | | AAM ³ | AAD ³ | SALC ³ | XLAT |
| E | LOOPNE/NZ Jb | LOOPE/Z Jb | LOOP Jb | JrcXZ Jb | IN AL, Ib eAX, Ib | | OUT Ib, AL Ib, eAX | |
| F | LOCK: | INT1 ICE Bkpt | REPNE: | REP: REPE: | HLT | CMC | Group 3 ² Eb | Group 3 ² Ev |

Note:

1. Rows in this table show the high opcode nibble, columns show the low opcode nibble.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.
4. Valid only in 64-bit mode.
5. Used as REX prefixes in 64-bit mode.

Table 4-2. One-Byte Opcodes, Low Nibble 8–Fh

| Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|---------------------|--|--------------------|-------------------------|----------------------|------------------------|----------------------|----------------------------------|---------------------------------------|
| 0 | OR Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb rAX, lz | | | | | | PUSH CS ³ | 2-byte opcodes |
| 1 | SBB Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb rAX, lz | | | | | | PUSH DS ³ | POP DS ³ |
| 2 | SUB Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb rAX, lz | | | | | | seg CS ³ | DAS ³ |
| 3 | CMP Eb, Gb Ev, Gv Gb, Eb Gv, Ev AL, lb rAX, lz | | | | | | seg DS ³ | AAS ³ |
| 4 | DEC ⁵ rAX rCX rDX rBX rSP rBP rSI rDI | | | | | | | |
| 5 | POP rAX/r8 rCX/r9 rDX/r10 rBX/r11 rSP/r12 rBP/r13 rSI/r14 rDI/r15 | | | | | | | |
| 6 | PUSH lz | IMUL Gv, Ev, lz | PUSH lb | IMUL Gv, Ev, lb | INSB Yb, DX | INSW/D Yv, DX | OUTSB DX, Xb | OUTSW/D DX, Xv |
| 7 | JS Jb | JNS Jb | JP Jb | JNP Jb | JL Jb | JNL Jb | JLE Jb | JNLE Jb |
| 8 | MOV Eb, Gb Ev, Gv Gb, Eb Gv, Ev Ew, Sw | | | | | LEA Gv, M | MOV Sw, Ew | Group 1a ² |
| 9 | CBW, CWDE CDQE | CWD, CDQ, CQO | CALL ³ Ap | WAIT FWAIT | PUSHF/D/Q Fv | POPF/D/Q Fv | SAHF ³ | LAHF ³ |
| A | TEST AL, lb rAX, lz | | STOSB Yb, AL | STOSW/D/Q Yv, rAX | LODSB AL, Xb | LODSW/D/Q rAX, Xv | SCASB AL, Yb | SCASW/D/Q rAX, Yv |
| B | MOV rAX, lv rCX, lv rDX, lv rBX, lv rSP, lv rBP, lv rSI, lv rDI, lv | | | | | | | |
| C | ENTER lw, lb | LEAVE | RET far lw | RET far | INT3 | INT lb | INTO ³ | IRET, IRETD IRETQ |
| D | x87 see Table 4-10 on page 401 | | | | | | | |
| E | CALL Jz | Jz | JMP Ap ³ | Jb | IN AL, DX eAX, DX | | OUT DX, AL DX, eAX | |
| F | CLC | STC | CLI | STI | CLD | STD | Group 4 ² INC, DEC | Group 5 ² INC, DEC etc. |

Note:

1. Rows in this table show the high opcode nibble, columns show the low opcode nibble.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.
4. Valid only in 64-bit mode.
5. Used as REX prefixes in 64-bit mode.

A.2.2 Two-Byte Opcodes

All two-byte opcodes have 0Fh as their first byte. Table 4-3 below shows the second byte of the two-byte opcodes in which the second byte's low nibble is in the range 0–7h. Table 4-4 on page 388 shows those opcodes in which the second byte's low nibble is in the range 8–Fh. In both tables, the rows show the full range (0–Fh) of the high nibble, and the columns show the low nibble of the opcode. The left-most column shows special-purpose prefix bytes used in many 128-bit and 64-bit instructions to modify the opcode.

Table 4-3. Second Byte of Two-Byte Opcodes, Low Nibble 0–7h

| Prefix | Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|--------|---------------------|--|----------------------|--------------------------------|--------------------|---------------------|---------------------|--------------------------------|--------------------|
| none | 0 | Group 6 ² | Group 7 ² | LAR Gv, Ew | LSL Gv, Ew | invalid | SYSCALL | CLTS | SYSRET |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 1 | MOVUPS Vps, Wps Wps, Vps | | MOVLPS, MOVHLPS Vps, Wps | MOVLPS Wps, Vps | UNPCKLPS Vps, Wq | UNPCKHPS Vps, Wq | MOVHPS, MOVLHPS Vps, Wps | MOVHPS Wps, Vps |
| F3 | | MOVSS Vss, Wss | MOVSS Wss, Vss | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | MOVUPD Vpd, Wpd | MOVUPD Wpd, Vpd | MOVLPD Vq, Mq | MOVLPD Mq, Vq | UNPCKLPD Vpd, Wq | UNPCKHPD Vpd, Wq | MOVHPD Vq, Mq | MOVHPD Mq, Vq |
| F2 | | MOVSD Vsd, Wsd | MOVSD Wsd, Vsd | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 2 | MOV Rd/q, Cd/q Rd/q, Dd/q Cd/q, Rd/q Dd/q, Rd/q | | | | invalid | invalid | invalid | invalid |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an 0Fh byte. Rows in the table show the high nibble of the second opcode bytes, columns show the low nibble of this byte.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.

Table 4-3. Second Byte of Two-Byte Opcodes, Low Nibble 0–7h (continued)

| Prefix | Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------------|---------------------|----------------------|----------------------|----------------------------|----------------------------|-----------------------|----------------------|----------------------|----------------------|
| none | 3 | WRMSR | RDTSR | RDMSR | RDPMC | SYSENTER ³ | SYSEXIT ³ | invalid | invalid |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 4 | CMOVO Gv, Ev | CMOVNO Gv, Ev | CMOVNB/C /NAE Gv, Ev | CMOVAE /NB/NC Gv, Ev | CMOVE/Z Gv, Ev | CMOVNE /NZ Gv, Ev | CMOVBE /NA Gv, Ev | CMOVA /NBE Gv, Ev |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 5 | MOVMSKPS Ed, Vps | SQRTPS Vps, Wps | RSQRTPS Vps, Wps | RCPPS Vps, Wps | ANDPS Vps, Wps | ANDNPS Vps, Wps | ORPS Vps, Wps | XORPS Vps, Wps |
| F3 | | invalid | SQRTSS Vss, Wss | RSQRTSS Vss, Wss | RCPPS Vss, Wss | invalid | invalid | invalid | invalid |
| 66 | | MOVMSKPD Ed, Vpd | SQRTPD Vpd, Wpd | invalid | invalid | ANDPD Vpd, Wpd | ANDNPD Vpd, Wpd | ORPD Vpd, Wpd | XORPD Vpd, Wpd |
| F2 | | invalid | SQRTSD Vsd, Wsd | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 6 | PUNPCKLBW Pq, Qd | PUNPCKLWD Pq, Qd | PUNPCKLDQ Pq, Qd | PACKSSWB Pq, Qq | PCMPGTB Pq, Qq | PCMPGTW Pq, Qq | PCMPGTD Pq, Qq | PACKUSWB Pq, Qq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | PUNPCKLBW Vdq, Wq | PUNPCKLWD Vdq, Wq | PUNPCKLDQ Vdq, Wq | PACKSSWB Vdq, Wdq | PCMPGTB Vdq, Wdq | PCMPGTW Vdq, Wdq | PCMPGTD Vdq, Wdq | PACKUSWB Vdq, Wdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an 0Fh byte. Rows in the table show the high nibble of the second opcode bytes, columns show the low nibble of this byte.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.

Table 4-3. Second Byte of Two-Byte Opcodes, Low Nibble 0–7h (continued)

| Prefix | Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------------|---------------------|------------------------|-----------------------|-----------------------|-----------------------|---------------------|---------------------|---------------------|--------------|
| none | 7 | PSHUFW Pq, Qq, Ib | Group 12 ² | Group 13 ² | Group 14 ² | PCMPEQB Pq, Qq | PCMPEQW Pq, Qq | PCMPEQD Pq, Qq | EMMS |
| F3 | | PSHUFW Vq, Wq, Ib | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | PSHUFD Vdq, Wdq, Ib | Group 12 ² | Group 13 ² | Group 14 ² | PCMPEQB Vdq, Wdq | PCMPEQW Vdq, Wdq | PCMPEQD Vdq, Wdq | invalid |
| F2 | | PSHUFLW Vq, Wq, Ib | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 8 | JO Jz | JNO Jz | JB Jz | JNB Jz | JZ Jz | JNZ Jz | JBE Jz | JNBE Jz |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 9 | SETO Eb | SETNO Eb | SETB Eb | SETNB Eb | SETZ Eb | SETNZ Eb | SETBE Eb | SETNBE Eb |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | A | PUSH FS | POP FS | CPUID | BT Ev, Gv | SHLD Ev, Gv, Ib | SHLD Ev, Gv, CL | invalid | invalid |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an 0Fh byte. Rows in the table show the high nibble of the second opcode bytes, columns show the low nibble of this byte.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.

Table 4-3. Second Byte of Two-Byte Opcodes, Low Nibble 0–7h (continued)

| Prefix | Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------------|---------------------|-------------------|-------------------|-----------------------|-------------------|-----------------------|-----------------------|------------------------|----------------------|
| none | B | CMPXCHG Eb, Gb | | LSS Gv, Mp | BTR Ev, Gv | LFS Gv, Mp | LGS Gv, Mp | MOVZX Gv, Eb | MOVZX Gv, Ew |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | C | XADD Eb, Gb | | CMPPS Vps, Wps, Ib | MOVNTI Md, Gd | PINSRW Pq, Ed, Ib | PEXTRW Gd, Pq, Ib | SHUFPS Vps, Wps, Ib | Group 9 ² |
| F3 | | invalid | invalid | CMPSS Vss, Wss, Ib | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | CMPPD Vpd, Wpd, Ib | invalid | PINSRW Vdq, Ed, Ib | PEXTRW Gd, Vdq, Ib | SHUFPD Vpd, Wpd, Ib | invalid |
| F2 | | invalid | invalid | CMPSD Vsd, Wsd, Ib | invalid | invalid | invalid | invalid | invalid |
| none | D | invalid | PSRLW Pq, Qq | PSRLD Pq, Qq | PSRLQ Pq, Qq | PADDQ Pq, Qq | PMULLW Pq, Qq | invalid | PMOVMASKB Gd, Pq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | MOVQ2DQ Vq, Pq | invalid |
| 66 | | invalid | PSRLW Vdq, Wdq | PSRLD Vdq, Wdq | PSRLQ Vdq, Wdq | PADDQ Vdq, Wdq | PMULLW Vdq, Wdq | MOVQ Wq, Vq | PMOVMASKB Gd, Vq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | MOVDQ2Q Pq, Vq | invalid |
| none | E | PAVGB Pq, Qq | PSRAW Pq, Qq | PSRAD Pq, Qq | PAVGW Pq, Qq | PMULHUW Pq, Qq | PMULHW Pq, Qq | invalid | MOVNTQ Mq, Pq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | CVTDQ2PD Vpd, Wq | invalid |
| 66 | | PAVGB Vdq, Wdq | PSRAW Vdq, Wdq | PSRAD Vdq, Wdq | PAVGW Vdq, Wdq | PMULHUW Vdq, Wdq | PMULHW Vdq, Wdq | CVTPD2DQ Vq, Wpd | MOVNTDQ Mdq, Vdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | CVTPD2DQ Vq, Wpd | invalid |

Note:

1. All two-byte opcodes begin with an *OFh* byte. Rows in the table show the high nibble of the second opcode bytes, columns show the low nibble of this byte.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.

Table 4-3. Second Byte of Two-Byte Opcodes, Low Nibble 0–7h (continued)

| Prefix | Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----------|---------------------|---------|-------------------|-------------------|-------------------|---------------------|---------------------|---------------------|-----------------------------|
| none | F | invalid | PSLLW Pq, Qq | PSLLD Pq, Qq | PSLLQ Pq, Qq | PMULUDQ Pq, Qq | PMADDWD Pq, Qq | PSADBWB Pq, Qq | MASKMOVQ Pq, Pq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | PSLLW Vdq, Wdq | PSLLD Vdq, Wdq | PSLLQ Vdq, Wdq | PMULUDQ Vdq, Wdq | PMADDWD Vdq, Wdq | PSADBWB Vdq, Wdq | MASK- MOVQDU Vdq, Vdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an *OFh* byte. Rows in the table show the high nibble of the second opcode bytes, columns show the low nibble of this byte.
2. An opcode extension is specified in bits 5–3 of the ModRM byte. See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.
3. Invalid in 64-bit mode.

Table 4-4. Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh

| Prefix | Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|-----------|---------------------|----------------------|---------|---------|---------|---------|--------------------------------------|---------|--|
| none | 0 | INVD | WBINVD | invalid | UD2 | invalid | PREFETCH Group P ² | FEMMS | 3DNow! See “3DNow!™ Opcodes” on page 397 |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 1 | Group P ² | NOP | NOP | NOP | NOP | NOP | NOP | NOP |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an *OFh* byte. Rows show high opcode nibble (hex), columns show low opcode nibble in hex.
2. An opcode extension is specified in the ModRM reg field (bits 5–3). See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.

Table 4-4. Second Byte of Two-Byte Opcodes, Low Nibble 8-Fh (continued)

| Prefix | Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|-------------|---------------------|----------------------------------|--------------------|----------------------|-----------------------|----------------------|---------------------|---------------------|---------------------|
| none | 2 | MOVAPS Vps, Wps Wps, Vps | | CVTPI2PS Vps, Qq | MOVNTPS Wps, Vps | CVTTPS2PI Pq, Wps | CVTPS2PI Pq, Wps | UCOMISS Vss, Wss | COMISS Vps, Wps |
| F3 | | invalid | invalid | CVTSI2SS Vss, Ed | invalid | CVTTSS2SI Gd, Wss | CVTSS2SI Gd, Wss | invalid | invalid |
| 66 | | MOVAPD Vpd, Wpd | MOVAPD Wpd, Vpd | CVTPI2PD Vpd, Qq | MOVNTPD Mpd, Vpd | CVTTPD2PI Qq, Wpd | CVTPD2PI Qq, Wpd | UCOMISD Vsd, Wsd | COMISD Vpd, Wsd |
| F2 | | invalid | invalid | CVTSI2SD Vsd, Ed | invalid | CVTTSD2SI Gd, Wsd | CVTSD2SI Gd, Wsd | invalid | invalid |
| none | 3 | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 4 | CMOVS Gv, Ev | CMOVNS Gv, Ev | CMOVP/PE Gv, Ev | CMOVNP/PO Gv, Ev | CMOVL/NGE Gv, Ev | CMOVGE/NL Gv, Ev | CMOVLE/NG Gv, Ev | CMOVG/NLE Gv, Ev |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 5 | ADDPS Vps, Wps | MULPS Vps, Wps | CVTPS2PD Vpd, Wps | CVTDQ2PS Vps, Wdq | SUBPS Vps, Wps | MINPS Vps, Wps | DIVPS Vps, Wps | MAXPS Vps, Wps |
| F3 | | ADDSS Vss, Wss | MULSS Vss, Wss | CVTSS2SD Vsd, Wss | CVTTPS2DQ Vdq, Wps | SUBSS Vss, Wss | MINSS Vss, Wss | DIVSS Vss, Wss | MAXSS Vss, Wss |
| 66 | | ADDPD Vpd, Wpd | MULPD Vpd, Wpd | CVTPD2PS Vps, Wpd | CVTPS2DQ Vdq, Wps | SUBPD Vpd, Wpd | MINPD Vpd, Wpd | DIVPD Vpd, Wpd | MAXPD Vpd, Wpd |
| F2 | | ADDSD Vsd, Wsd | MULSD Vsd, Wsd | CVTSD2SS Vss, Wsd | invalid | SUBSD Vsd, Wsd | MINSD Vsd, Wsd | DIVSD Vsd, Wsd | MAXSD Vsd, Wsd |

Note:

1. All two-byte opcodes begin with an OFh byte. Rows show high opcode nibble (hex), columns show low opcode nibble in hex.
2. An opcode extension is specified in the ModRM reg field (bits 5–3). See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.

Table 4-4. Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh (continued)

| Prefix | Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|-------------|---------------------|-----------------------|-----------------------|-----------------------|----------------------|------------------------|------------------------|----------------|--------------------|
| none | 6 | PUNPCK-HBW Pq, Qd | PUNPCK-HWD Pq, Qd | PUNPCK-HDQ Pq, Qd | PACKSSDW Pq, Qq | invalid | invalid | MOVD Pq, Ed | MOVQ Pq, Qq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | MOVDQU Vdq, Wdq |
| 66 | | PUNPCK-HBW Vdq, Wq | PUNPCK-HWD Vdq, Wq | PUNPCK-HDQ Vdq, Wq | PACKSSDW Vdq, Wdq | PUNPCK-LQDQ Vdq, Wq | PUNPCK-HQDQ Vdq, Wq | MOVD Vq, Ed | MOVDQA Vdq, Wdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 7 | invalid | invalid | invalid | invalid | invalid | invalid | MOVD Ed, Pd | MOVQ Qq, Pq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | MOVQ Vq, Wq | MOVDQU Wdq, Vdq |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | MOVD Ed, Vd | MOVDQA Wdq, Vdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 8 | JS Jz | JNS Jz | JP Jz | JNP Jz | JL Jz | JNL Jz | JLE Jz | JNLE Jz |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | 9 | SETS Eb | SETNS Eb | SETP Eb | SETNP Eb | SETL Eb | SETNL Eb | SETLE Eb | SETNLE Eb |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an OFh byte. Rows show high opcode nibble (hex), columns show low opcode nibble in hex.
2. An opcode extension is specified in the ModRM reg field (bits 5–3). See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.

Table 4-4. Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh (continued)

| Prefix | Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|-------------|---------------------|---------------------|------------------------------|--------------------------------|----------------------|----------------------|----------------------|-----------------------|----------------------|
| none | A | PUSH GS | POP GS | RSM | BTS Ev, Gv | SHRD Ev, Gv, Ib | SHRD Ev, Gv, CL | Group 15 ² | IMUL Gv, Ev |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | B | invalid | Group 10 ² UD2 | Group 8 ² Ev, Ib | BTC Ev, Gv | BSF Gv, Ev | BSR Gv, Ev | MOVSX Gv, Eb | MOVSX Gv, Ew |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | C | BSWAP EAX/RAX/R8 | BSWAP ECX/RCX/R9 | BSWAP EDX/RDX/R10 | BSWAP EBX/RBX/R11 | BSWAP ESP/RSP/R12 | BSWAP EBP/RBP/R13 | BSWAP ESI/RSI/R14 | BSWAP EDI/RDI/R15 |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | D | PSUBUSB Pq, Qq | PSUBUSW Pq, Qq | PMINUB Pq, Qq | PAND Pq, Qq | PADDUSB Pq, Qq | PADDUSW Pq, Qq | PMAXUB Pq, Qq | PANDN Pq, Qq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | PSUBUSB Vdq, Wdq | PSUBUSW Vdq, Wdq | PMINUB Vdq, Wdq | PAND Vdq, Wdq | PADDUSB Vdq, Wdq | PADDUSW Vdq, Wdq | PMAXUB Vdq, Wdq | PANDN Vdq, Wdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an OFh byte. Rows show high opcode nibble (hex), columns show low opcode nibble in hex.
2. An opcode extension is specified in the ModRM reg field (bits 5–3). See “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 for details.

Table 4-4. Second Byte of Two-Byte Opcodes, Low Nibble 8–Fh (continued)

| Prefix | Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|-------------|---------------------|--------------------|--------------------|--------------------|-------------------|--------------------|--------------------|--------------------|------------------|
| none | E | PSUBSB Pq, Qq | PSUBSW Pq, Qq | PMINSW Pq, Qq | POR Pq, Qq | PADDSB Pq, Qq | PADDSW Pq, Qq | PMAXSW Pq, Qq | PXOR Pq, Qq |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | PSUBSB Vdq, Wdq | PSUBSW Vdq, Wdq | PMINSW Vdq, Wdq | POR Vdq, Wdq | PADDSB Vdq, Wdq | PADDSW Vdq, Wdq | PMAXSW Vdq, Wdq | PXOR Vdq, Wdq |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| none | F | PSUBB Pq, Qq | PSUBW Pq, Qq | PSUBD Pq, Qq | PSUBQ Pq, Qq | PADDB Pq, Qq | PADDW Pq, Qq | PADD Pq, Qq | invalid |
| F3 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 66 | | PSUBB Vdq, Wdq | PSUBW Vdq, Wdq | PSUBD Vdq, Wdq | PSUBQ Vdq, Wdq | PADDB Vdq, Wdq | PADDW Vdq, Wdq | PADD Vdq, Wdq | invalid |
| F2 | | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. All two-byte opcodes begin with an *OFh* byte. Rows show high opcode nibble (hex), columns show low opcode nibble in hex.
2. An opcode extension is specified in the *ModRM reg* field (bits 5–3). See “*ModRM Extensions to One-Byte and Two-Byte Opcodes*” on page 394 for details.

A.2.3 rFLAGS Condition Codes for Two-Byte Opcodes

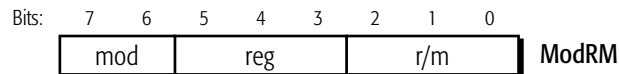
Table 4-5 shows the rFLAGS condition codes specified by the low nibble in the second opcode byte of the CMOVcc, Jcc, and SETcc instructions.

Table 4-5. rFLAGS Condition Codes for CMOVcc, Jcc, and SETcc

| Low Nibble of Second Opcode Byte (hex) | rFLAGS Value | cc Mnemonic | Arithmetic Type | Condition(s) |
|--|---------------------------|-------------|-----------------|---|
| 0 | OF = 1 | O | | Overflow |
| 1 | OF = 0 | NO | | No Overflow |
| 2 | CF = 1 | B, C, NAE | Unsigned | Below, Carry, Not Above or Equal |
| 3 | CF = 0 | NB, NC, AE | | Not Below, No Carry, Above or Equal |
| 4 | ZF = 1 | E, Z | | Equal, Zero |
| 5 | ZF = 0 | NE, NZ | | Not Equal, Not Zero |
| 6 | CF = 1 or ZF = 1 | BE, NA | | Below or Equal, Not Above |
| 7 | CF = 0 or ZF = 0 | NBE, A | | Not Below or Equal, Above |
| 8 | SF = 1 | S | | Sign |
| 9 | SF = 0 | NS | | Not Sign |
| A | PF = 1 | P, PE, U | Signed | Parity, Parity Even, Unordered |
| B | PF = 0 | NP, PO, NU | | Not Parity, Parity Odd, Not Unordered |
| C | (SF xor OF) = 1 | L, NGE | | Less than, Not Greater than or Equal to |
| D | (SF xor OF) = 0 | NL, GE | | Not Less than, Greater than or Equal to |
| E | (SF xor OF) = 1 or ZF = 1 | LE, NG | | Less than or Equal to, Not Greater than |
| F | (SF xor OF) = 0 or ZF = 0 | NLE, G | | Not Less than or Equal to, Greater than |

A.2.4 ModRM Extensions to One-Byte and Two-Byte Opcodes

The ModRM byte, which immediately follows the last opcode byte, is used in certain instruction encodings to provide additional opcode bits with which to define the function of the instruction. ModRM bytes have three fields—*mod*, *reg*, and *r/m*, as shown in Figure 4-2.



513-325.eps

Figure 4-1. ModRM-Byte *reg* Field

In most cases, the *reg* field (bits 5–3) provides the additional bits with which to extend the encodings of the first one or two opcode bytes. In the case of the 64-bit media instructions and many of the x87 floating-point instructions, the entire ModRM byte is used to extend the opcode encodings.

Table 4-6 shows how the ModRM *reg* field is used to extend the range of one-byte and two-byte opcodes. In all cases but two, the binary value in the ModRM *mod* field (bits 7–6) is 11. (The two exceptions are CLFLUSH and INVLPG). The opcode ranges are organized into *groups* of opcode extensions. The group number is shown in the left-most column of Table 4-6. These groups are referenced in the opcodes shown in Table 4-1 on page 381 through Table 4-4 on page 388. An entry of “n.a.” in the Prefix column means that prefixes are not applicable to the opcodes in that row. Prefixes only apply to certain 128-bit media, 64-bit media, and a few other instructions introduced with the SSE or SSE2 technologies.

The /0 through /7 notation for the ModRM *reg* field (bits 5–3) means that the three-bit field contains a value from zero (binary 000) to 7 (binary 111).

Table 4-6. One-Byte and Two-Byte Opcode ModRM Extensions

| Group Number | Opcode | Prefix | ModRM <i>reg</i> Field | | | | | | | |
|-----------------|--------|--------|----------------------------|---------------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|----------------------------|
| | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| Group 1 | 80 | n.a. | ADD Eb, Ib | OR Eb, Ib | ADC Eb, Ib | SBB Eb, Ib | AND Eb, Ib | SUB Eb, Ib | XOR Eb, Ib | CMP Eb, Ib |
| | 81 | | ADD Ev, Iz | OR Ev, Iz | ADC Ev, Iz | SBB Ev, Iz | AND Ev, Iz | SUB Ev, Iz | XOR Ev, Iz | CMP Ev, Iz |
| | 82 | | ADD Eb, Ib ² | OR Eb, Ib ² | ADC Eb, Ib ² | SBB Eb, Ib ² | AND Eb, Ib ² | SUB Eb, Ib ² | XOR Eb, Ib ² | CMP Eb, Ib ² |
| | 83 | | ADD Ev, Ib | OR Ev, Ib | ADC Ev, Ib | SBB Ev, Ib | AND Ev, Ib | SUB Ev, Ib | XOR Ev, Ib | CMP Ev, Ib |
| Group 1a | 8F | n.a. | POP Ev | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 2 | C0 | n.a. | ROL Eb, Ib | ROR Eb, Ib | RCL Eb, Ib | RCR Eb, Ib | SHL/SAL Eb, Ib | SHR Eb, Ib | SHL/SAL Eb, Ib | SAR Eb, Ib |
| | C1 | | ROL Ev, Ib | ROR Ev, Ib | RCL Ev, Ib | RCR Ev, Ib | SHL/SAL Ev, Ib | SHR Ev, Ib | SHL/SAL Ev, Ib | SAR Ev, Ib |
| | D0 | | ROL Eb, 1 | ROR Eb, 1 | RCL Eb, 1 | RCR Eb, 1 | SHL/SAL Eb, 1 | SHR Eb, 1 | SHL/SAL Eb, 1 | SAR Eb, 1 |
| | D1 | | ROL Ev, 1 | ROR Ev, 1 | RCL Ev, 1 | RCR Ev, 1 | SHL/SAL Ev, 1 | SHR Ev, 1 | SHL/SAL Ev, 1 | SAR Ev, 1 |
| | D2 | | ROL Eb, CL | ROR Eb, CL | RCL Eb, CL | RCR Eb, CL | SHL/SAL Eb, CL | SHR Eb, CL | SHL/SAL Eb, CL | SAR Eb, CL |
| | D3 | | ROL Ev, CL | ROR Ev, CL | RCL Ev, CL | RCR Ev, CL | SHL/SAL Ev, CL | SHR Ev, CL | SHL/SAL Ev, CL | SAR Ev, CL |
| Group 3 | F6 | n.a. | TEST Ib | | NOT Eb | NEG Eb | MUL Eb | IMUL AL | DIV AL | IDIV AL |
| | F7 | | TEST Iz | | NOT Ev | NEG Ev | MUL rAX | IMUL rAX | DIV rAX | IDIV rAX |
| Group 4 | FE | n.a. | INC Eb | DEC Eb | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 5 | FF | n.a. | INC Ev | DEC Ev | CALL Ev | CALL Ep | JMP Ev | JMP Ep | PUSH Ev | invalid |
| Group 6 | 0F 00 | n.a. | SLDT Ew | STR Ew | LLDT Ew | LTR Ew | VERR Ew | VERW Ew | invalid | invalid |

Note:

1. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode SWAPGS.
2. Invalid in 64-bit mode.
3. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode SFENCE.
4. This instruction takes a ModRM byte.
5. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) if otherwise undefined.

Table 4-6. One-Byte and Two-Byte Opcode ModRM Extensions (continued)

| Group Number | Opcode | Prefix | ModRM <i>reg</i> Field | | | | | | | |
|-----------------|--------|---------------|------------------------|---------------|------------------|-------------------|------------------|---------|------------------|--------------------------------------|
| | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| Group 7 | 0F 01 | n.a. | SGDT Ms | SIDT Ms | LGDT Ms | LIDT Ms | SMSW Ew | invalid | LMSW Ew | INVLPG Mb SWAPGS ¹ |
| Group 8 | 0F BA | n.a. | invalid | invalid | invalid | invalid | BT | BTS | BTR | BTC |
| Group 9 | 0F C7 | n.a. | invalid | CMPXCHG Mq | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 10 | 0F B9 | n.a. | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 11 | C6:C7 | n.a. | MOV | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 12 | 0F 71 | none | invalid | invalid | PSRLW Pq, Ib | invalid | PSRAW Pq, Ib | invalid | PSLLW Pq, Ib | invalid |
| | | 66 | invalid | invalid | PSRLW Vdq, Ib | invalid | PSRAW Vdq, Ib | invalid | PSLLW Vdq, Ib | invalid |
| | | F2, F3 | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 13 | 0F 72 | none | invalid | invalid | PSRLD Pq, Ib | invalid | PSRAD Pq, Ib | invalid | PSLLD Pq, Ib | invalid |
| | | 66 | invalid | invalid | PSRLD Vdq, Ib | invalid | PSRAD Vdq, Ib | invalid | PSLLD Vdq, Ib | invalid |
| | | F2, F3 | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 14 | 0F 73 | none | invalid | invalid | PSRLQ Pq, Ib | invalid | invalid | invalid | PSLLQ Pq, Ib | invalid |
| | | 66 | invalid | invalid | PSRLQ Vdq, Ib | PSRLDQ Vdq, Ib | invalid | invalid | PSLLQ Vdq, Ib | PSLLDQ Vdq, Ib |
| | | F2, F3 | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| Group 15 | 0F AE | none | FXSAVE | FXRSTOR | LDMXCSR | STMXCSR | invalid | LFENCE | MFENCE | SFENCE ³ CLFLUSH Mb |
| | | 66, F2, F3 | invalid | invalid | invalid | invalid | invalid | invalid | invalid | invalid |

Note:

1. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode SWAPGS.
2. Invalid in 64-bit mode.
3. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode SFENCE.
4. This instruction takes a ModRM byte.
5. Reserved prefetch encodings are aliased to the /0 encoding (PREFETCH Exclusive) if otherwise undefined.

Table 4-6. One-Byte and Two-Byte Opcode ModRM Extensions (continued)

| Group Number | Opcode | Prefix | ModRM <i>reg</i> Field | | | | | | | |
|-----------------|--------|--------|------------------------|----------------------|-----------------------------------|----------------------|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|
| | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| Group 16 | 0F 18 | n.a. | PREFETCH NTA | PREFETCH T0 | PREFETCH T1 | PREFETCH T2 | NOP ⁴ | NOP ⁴ | NOP ⁴ | NOP ⁴ |
| Group P | 0F 0D | n.a. | PREFETCH Exclusive | PREFETCH Modified | Prefetch Reserved ⁵ | PREFETCH Modified | Prefetch Reserved ⁵ | Prefetch Reserved ⁵ | Prefetch Reserved ⁵ | Prefetch Reserved ⁵ |

Note:

1. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode *SWAPGS*.
2. Invalid in 64-bit mode.
3. See Table 4-7 on page 397 for ModRM extensions of this two-byte opcode to encode *SFENCE*.
4. This instruction takes a ModRM byte.
5. Reserved prefetch encodings are aliased to the /0 encoding (*PREFETCH Exclusive*) if otherwise undefined.

A.2.5 ModRM Extensions to *SWAPGS* and *CLFLUSH* Opcodes

Table 4-7 shows the ModRM *r/m* field encodings for the 0F 01 and 0F AE opcodes, shown in Table 4-6. The 0F 01 /7 opcode is shared by the *INVLPG* and *SWAPGS* instructions and the 0F AE /7 opcode is shared by the *SFENCE* and *CLFLUSH* instructions. The opcodes are differentiated by the fact that the binary value of the ModRM *mod* field is always 11 for *SWAPGS* and *SFENCE*, and any value except 11 for *INVLPG* and *CLFLUSH*. The *SWAPGS* opcode is only valid in 64-bit mode.

Table 4-7. *SWAPGS* and *CLFLUSH* ModRM Extensions

| Opcode | ModRM <i>r/m</i> Field | | | | | | | |
|--------------------|------------------------|---------|---------|---------|---------|---------|---------|---------|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0F 01 /7 mod=11 | <i>SWAPGS</i> | invalid | invalid | invalid | invalid | invalid | invalid | invalid |
| 0F AE /7 mod=11 | <i>SFENCE</i> | | | | | | | |

A.2.6 3DNow!™ Opcodes

The 64-bit media instructions include the MMX™ instructions and the AMD 3DNow! instructions. The MMX instructions are encoded using two opcode bytes, as described in “Two-Byte Opcodes” on page 384.

The 3DNow! instructions are encoded using two 0Fh opcode bytes and an immediate byte that is located at the last byte position of the instruction encoding. Thus, the format for 3DNow! instructions is:

0Fh 0Fh [ModRM] [SIB] [displacement] *imm8_opcode*

Table 4-8 and Table 4-9 show the immediate byte following the opcode bytes for 3DNow! instructions. In these tables, rows show the high nibble of the immediate byte, and columns show the low nibble of the immediate byte. Table 4-8 shows the immediate bytes whose low nibble is in the range 0–7h. Table 4-9 shows the same for immediate bytes whose low nibble is in the range 8–Fh.

Byte values shown as *reserved* in these tables have implementation-specific functions, which can include an invalid-opcode exception.

Table 4-8. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 0–7h

| Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------------------|-------------------|----------|----------|----------|-------------------|----------|--------------------|--------------------|
| 0 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 2 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 3 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 4 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 5 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 6 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 7 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 8 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 9 | PFCMPGE Pq, Qq | reserved | reserved | reserved | PFCMPGE Pq, Qq | reserved | PFCMPGE Pq, Qq | PFCMPGE Pq, Qq |
| A | PFCMPGT Pq, Qq | reserved | reserved | reserved | PFCMPGT Pq, Qq | reserved | PFCMPGT1 Pq, Qq | PFCMPGT1 Pq, Qq |
| B | PFCMPEQ Pq, Qq | reserved | reserved | reserved | PFCMPEQ Pq, Qq | reserved | PFCMPEQ2 Pq, Qq | PFCMPEQ2 Pq, Qq |
| C | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |

Note:

1. All 3DNow! opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

Table 4-8. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 0–7h (continued)

| Nibble ¹ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------------------|----------|----------|----------|----------|----------|----------|----------|----------|
| D | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| E | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| F | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |

Note:

1. All 3DNow! opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

Table 4-9. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 8–Fh

| Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|---------------------|----------|----------|------------------|------------------|-----------------|-----------------|-------------------|-------------------|
| 0 | reserved | reserved | reserved | reserved | PI2FW Pq, Qq | PI2FD Pq, Qq | reserved | reserved |
| 2 | reserved | reserved | reserved | reserved | PF2IW Pq, Qq | PF2ID Pq, Qq | reserved | reserved |
| 3 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 4 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 5 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 6 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 7 | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| 8 | reserved | reserved | PFNACC Pq, Qq | reserved | reserved | reserved | PFPNACC Pq, Qq | reserved |
| 9 | reserved | reserved | PFSUB Pq, Qq | reserved | reserved | reserved | PFADD Pq, Qq | reserved |
| A | reserved | reserved | PFSUBR Pq, Qq | reserved | reserved | reserved | PFACC Pq, Qq | reserved |
| B | reserved | reserved | reserved | PSWAPD Pq, Qq | reserved | reserved | reserved | PAVGUSB Pq, Qq |
| C | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |

Note:

1. All 3DNow! opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

Table 4-9. Immediate Byte for 3DNow!™ Opcodes, Low Nibble 8–Fh (continued)

| Nibble ¹ | 8 | 9 | A | B | C | D | E | F |
|---------------------|----------|----------|----------|----------|----------|----------|----------|----------|
| D | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| E | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |
| F | reserved | reserved | reserved | reserved | reserved | reserved | reserved | reserved |

Note:

1. All 3DNow! opcodes consist of two 0Fh bytes. This table shows the immediate byte for 3DNow! opcodes. Rows show the high nibble of the immediate byte. Columns show the low nibble of the immediate byte.

A.2.7 x87 Encodings

All x87 instructions begin with an opcode byte in the range D8h to DFh, as shown in Table 4-2 on page 383. These opcodes are followed by a ModRM byte that further defines the opcode. Table 4-10 shows both the opcode byte and the ModRM byte for each x87 instruction.

There are two significant ranges of the ModRM *reg* field (bits 5–3) for x87 opcodes: 00–BFh and C0–FFh. When the value of the ModRM *reg* field falls within the first range, 00–BFh, the opcode uses only this *reg* field to further define the opcode. When the value of the *reg* field falls within the second range, C0–FFh, the opcode uses the entire ModRM byte to further define the opcode.

Byte values shown as *reserved* in Table 4-10 have implementation-specific functions, which can include an invalid-opcode exception.

Table 4-10. x87 Opcodes and ModRM Extensions

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|-----------|------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|------------------------------------|-----------------------------------|------------------------------------|-----------------------------------|------------------------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| D8 | !11 | 00-BF | | | | | | | |
| | | FADD mem32real | FMUL mem32real | FCOM mem32real | FCOMP mem32real | FSUB mem32real | FSUBR mem32real | FDIV mem32real | FDIVR mem32real |
| | 11 | C0 FADD ST(0), ST(0) | C8 FMUL ST(0), ST(0) | D0 FCOM ST(0), ST(0) | D8 FCOMP ST(0), ST(0) | E0 FSUB ST(0), ST(0) | E8 FSUBR ST(0), ST(0) | F0 FDIV ST(0), ST(0) | F8 FDIVR ST(0), ST(0) |
| | | C1 FADD ST(0), ST(1) | C9 FMUL ST(0), ST(1) | D1 FCOM ST(0), ST(1) | D9 FCOMP ST(0), ST(1) | E1 FSUB ST(0), ST(1) | E9 FSUBR ST(0), ST(1) | F1 FDIV ST(0), ST(1) | F9 FDIVR ST(0), ST(1) |
| | | C2 FADD ST(0), ST(2) | CA FMUL ST(0), ST(2) | D2 FCOM ST(0), ST(2) | DA FCOMP ST(0), ST(2) | E2 FSUB ST(0), ST(2) | EA FSUBR ST(0), ST(2) | F2 FDIV ST(0), ST(2) | FA FDIVR ST(0), ST(2) |
| | | C3 FADD ST(0), ST(3) | CB FMUL ST(0), ST(3) | D3 FCOM ST(0), ST(3) | DB FCOMP ST(0), ST(3) | E3 FSUB ST(0), ST(3) | EB FSUBR ST(0), ST(3) | F3 FDIV ST(0), ST(3) | FB FDIVR ST(0), ST(3) |
| | | C4 FADD ST(0), ST(4) | CC FMUL ST(0), ST(4) | D4 FCOM ST(0), ST(4) | DC FCOMP ST(0), ST(4) | E4 FSUB ST(0), ST(4) | EC FSUBR ST(0), ST(4) | F4 FDIV ST(0), ST(4) | FC FDIVR ST(0), ST(4) |
| | | C5 FADD ST(0), ST(5) | CD FMUL ST(0), ST(5) | D5 FCOM ST(0), ST(5) | DD FCOMP ST(0), ST(5) | E5 FSUB ST(0), ST(5) | ED FSUBR ST(0), ST(5) | F5 FDIV ST(0), ST(5) | FD FDIVR ST(0), ST(5) |
| | | C6 FADD ST(0), ST(6) | CE FMUL ST(0), ST(6) | D6 FCOM ST(0), ST(6) | DE FCOMP ST(0), ST(6) | E6 FSUB ST(0), ST(6) | EE FSUBR ST(0), ST(6) | F6 FDIV ST(0), ST(6) | FE FDIVR ST(0), ST(6) |
| | | C7 FADD ST(0), ST(7) | CF FMUL ST(0), ST(7) | D7 FCOM ST(0), ST(7) | DF FCOMP ST(0), ST(7) | E7 FSUB ST(0), ST(7) | EF FSUBR ST(0), ST(7) | F7 FDIV ST(0), ST(7) | FF FDIVR ST(0), ST(7) |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|-----------|------------------------------|----------------------------------|-----------------------------------|----------------------|----------------------|---------------------------|----------------------|---------------------------|----------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| D9 | !11 | 00–BF | | | | | | | |
| | | FLD mem32real | invalid | FST mem32real | FSTP mem32real | FLDENV mem14/28e nv | FLDCW mem16 | FSTENV mem14/28e nv | FSTCW mem16 |
| | 11 | C0 FLD ST(0), ST(0) | C8 FXCH ST(0), ST(0) | D0 FNOP | D8 invalid | E0 FCHS | E8 FLD1 | F0 F2XM1 | F8 FPREM |
| | | C1 FLD ST(0), ST(1) | C9 FXCH ST(0), ST(1) | D1 invalid | D9 invalid | E1 FABS | E9 FLDL2T | F1 FYL2X | F9 FYL2XP1 |
| | | C2 FLD ST(0), ST(2) | CA FXCH ST(0), ST(2) | D2 invalid | DA invalid | E2 invalid | EA FLDL2E | F2 FPTAN | FA FSQRT |
| | | C3 FLD ST(0), ST(3) | CB FXCH ST(0), ST(3) | D3 invalid | DB invalid | E3 invalid | EB FLDPI | F3 FPATAN | FB FSINCOS |
| | | C4 FLD ST(0), ST(4) | CC FXCH ST(0), ST(4) | D4 invalid | DC invalid | E4 FTST | EC FLDLG2 | F4 EXTRACT | FC FRNDINT |
| | | C5 FLD ST(0), ST(5) | CD FXCH ST(0), ST(5) | D5 invalid | DD invalid | E5 FXAM | ED FLDLN2 | F5 FPREM1 | FD FSCALE |
| | | C6 FLD ST(0), ST(6) | CE FXCH ST(0), ST(6) | D6 invalid | DE invalid | E6 invalid | EE FLDZ | F6 FDECSTP | FE FSIN |
| | | C7 FLD ST(0), ST(7) | CF FXCH ST(0), ST(7) | D7 invalid | DF invalid | E7 invalid | EF invalid | F7 FINCSTP | FF FCOS |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|--------|------------------------------|-------------------------------------|-------------------------------------|--------------------------------------|-------------------------------------|----------------------|----------------------|----------------------|----------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DA | !11 | 00–BF | | | | | | | |
| | | FIADD mem32int | FIMUL mem32int | FICOM mem32int | FICOMP mem32int | FISUB mem32int | FISUBR mem32int | FIDIV mem32int | FIDIVR mem32int |
| | 11 | C0 FCMOVB ST(0), ST(0) | C8 FCMOVE ST(0), ST(0) | D0 FCMOVBE ST(0), ST(0) | D8 FCMOVU ST(0), ST(0) | E0 invalid | E8 invalid | F0 invalid | F8 invalid |
| | | C1 FCMOVB ST(0), ST(1) | C9 FCMOVE ST(0), ST(1) | D1 FCMOVBE ST(0), ST(1) | D9 FCMOVU ST(0), ST(1) | E1 invalid | E9 FUCOMPP | F1 invalid | F9 invalid |
| | | C2 FCMOVB ST(0), ST(2) | CA FCMOVE ST(0), ST(2) | D2 FCMOVBE ST(0), ST(2) | DA FCMOVU ST(0), ST(2) | E2 invalid | EA invalid | F2 invalid | FA invalid |
| | | C3 FCMOVB ST(0), ST(3) | CB FCMOVE ST(0), ST(3) | D3 FCMOVBE ST(0), ST(3) | DB FCMOVU ST(0), ST(3) | E3 invalid | EB invalid | F3 invalid | FB invalid |
| | | C4 FCMOVB ST(0), ST(4) | CC FCMOVE ST(0), ST(4) | D4 FCMOVBE ST(0), ST(4) | DC FCMOVU ST(0), ST(4) | E4 invalid | EC invalid | F4 invalid | FC invalid |
| | | C5 FCMOVB ST(0), ST(5) | CD FCMOVE ST(0), ST(5) | D5 FCMOVBE ST(0), ST(5) | DD FCMOVU ST(0), ST(5) | E5 invalid | ED invalid | F5 invalid | FD invalid |
| | | C6 FCMOVB ST(0), ST(6) | CE FCMOVE ST(0), ST(6) | D6 FCMOVBE ST(0), ST(6) | DE FCMOVU ST(0), ST(6) | E6 invalid | EE invalid | F6 invalid | FE invalid |
| | | C7 FCMOVB ST(0), ST(7) | CF FCMOVE ST(0), ST(7) | D7 FCMOVBE ST(0), ST(7) | DF FCMOVU ST(0), ST(7) | E7 invalid | EF invalid | F7 invalid | FF invalid |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|--------|------------------------------|--------------------------------------|--------------------------------------|---------------------------------------|--------------------------------------|----------------------|-------------------------------------|------------------------------------|----------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DB | !11 | 00-BF | | | | | | | |
| | | FILD mem32int | invalid | FIST mem32int | FISTP mem32int | invalid | FLD mem80real | invalid | FSTP mem80real |
| | 11 | C0 FCMOVNB ST(0), ST(0) | C8 FCMOVNE ST(0), ST(0) | D0 FCMOVNBE ST(0), ST(0) | D8 FCMOVNU ST(0), ST(0) | E0 invalid | E8 FUCOMI ST(0), ST(0) | F0 FCOMI ST(0), ST(0) | F8 invalid |
| | | C1 FCMOVNB ST(0), ST(1) | C9 FCMOVNE ST(0), ST(1) | D1 FCMOVNBE ST(0), ST(1) | D9 FCMOVNU ST(0), ST(1) | E1 invalid | E9 FUCOMI ST(0), ST(1) | F1 FCOMI ST(0), ST(1) | F9 invalid |
| | | C2 FCMOVNB ST(0), ST(2) | CA FCMOVNE ST(0), ST(2) | D2 FCMOVNBE ST(0), ST(2) | DA FCMOVNU ST(0), ST(2) | E2 FCLEX | EA FUCOMI ST(0), ST(2) | F2 FCOMI ST(0), ST(2) | FA invalid |
| | | C3 FCMOVNB ST(0), ST(3) | CB FCMOVNE ST(0), ST(3) | D3 FCMOVNBE ST(0), ST(3) | DB FCMOVNU ST(0), ST(3) | E3 FINIT | EB FUCOMI ST(0), ST(3) | F3 FCOMI ST(0), ST(3) | FB invalid |
| | | C4 FCMOVNB ST(0), ST(4) | CC FCMOVNE ST(0), ST(4) | D4 FCMOVNBE ST(0), ST(4) | DC FCMOVNU ST(0), ST(4) | E4 invalid | EC FUCOMI ST(0), ST(4) | F4 FCOMI ST(0), ST(4) | FC invalid |
| | | C5 FCMOVNB ST(0), ST(5) | CD FCMOVNE ST(0), ST(5) | D5 FCMOVNBE ST(0), ST(5) | DD FCMOVNU ST(0), ST(5) | E5 invalid | ED FUCOMI ST(0), ST(5) | F5 FCOMI ST(0), ST(5) | FD invalid |
| | | C6 FCMOVNB ST(0), ST(6) | CE FCMOVNE ST(0), ST(6) | D6 FCMOVNBE ST(0), ST(6) | DE FCMOVNU ST(0), ST(6) | E6 invalid | EE FUCOMI ST(0), ST(6) | F6 FCOMI ST(0), ST(6) | FE invalid |
| | | C7 FCMOVNB ST(0), ST(7) | CF FCMOVNE ST(0), ST(7) | D7 FCMOVNBE ST(0), ST(7) | DF FCMOVNU ST(0), ST(7) | E7 invalid | EF FUCOMI ST(0), ST(7) | F7 FCOMI ST(0), ST(7) | FF invalid |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|--------|------------------------------|-----------------------------------|-----------------------------------|----------------------|----------------------|------------------------------------|-----------------------------------|------------------------------------|-----------------------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DC | !11 | 00-BF | | | | | | | |
| | | FADD m64real | FMUL m64real | FCOM m64real | FCOMP m64real | FSUB m64real | FSUBR m64real | FDIV m64real | FDIVR m64real |
| | 11 | C0 FADD ST(0), ST(0) | C8 FMUL ST(0), ST(0) | D0 invalid | D8 invalid | E0 FSUBR ST(0), ST(0) | E8 FSUB ST(0), ST(0) | F0 FDIVR ST(0), ST(0) | F8 FDIV ST(0), ST(0) |
| | | C1 FADD ST(1), ST(0) | C9 FMUL ST(1), ST(0) | D1 | D9 | E1 FSUBR ST(1), ST(0) | E9 FSUB ST(1), ST(0) | F1 FDIVR ST(1), ST(0) | F9 FDIV ST(1), ST(0) |
| | | C2 FADD ST(2), ST(0) | CA FMUL ST(2), ST(0) | D2 invalid | DA invalid | E2 FSUBR ST(2), ST(0) | EA FSUB ST(2), ST(0) | F2 FDIVR ST(2), ST(0) | FA FDIV ST(2), ST(0) |
| | | C3 FADD ST(3), ST(0) | CB FMUL ST(3), ST(0) | D3 invalid | DB invalid | E3 FSUBR ST(3), ST(0) | EB FSUB ST(3), ST(0) | F3 FDIVR ST(3), ST(0) | FB FDIV ST(3), ST(0) |
| | | C4 FADD ST(4), ST(0) | CC FMUL ST(4), ST(0) | D4 invalid | DC invalid | E4 FSUBR ST(4), ST(0) | EC FSUB ST(4), ST(0) | F4 FDIVR ST(4), ST(0) | FC FDIV ST(4), ST(0) |
| | | C5 FADD ST(5), ST(0) | CD FMUL ST(5), ST(0) | D5 invalid | DD invalid | E5 FSUBR ST(5), ST(0) | ED FSUB ST(5), ST(0) | F5 FDIVR ST(5), ST(0) | FD FDIV ST(5), ST(0) |
| | | C6 FADD ST(6), ST(0) | CE FMUL ST(6), ST(0) | D6 invalid | DE invalid | E6 FSUBR ST(6), ST(0) | EE FSUB ST(6), ST(0) | F6 FDIVR ST(6), ST(0) | FE FDIV ST(6), ST(0) |
| | | C7 FADD ST(7), ST(0) | CF FMUL ST(7), ST(0) | D7 invalid | DF invalid | E7 FSUBR ST(7), ST(0) | EF FSUB ST(7), ST(0) | F7 FDIVR ST(7), ST(0) | FF FDIV ST(7), ST(0) |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|--------|------------------------------|-----------------------------|----------------------|---------------------------|----------------------------|------------------------------------|------------------------------|---------------------------|----------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DD | !11 | 00-BF | | | | | | | |
| | | FLD m64real | invalid | FST m64real | FSTP m64real | FRSTOR mem98/108 env | invalid | FSAVE mem98/108 env | FSTSW mem16 |
| | 11 | C0 FFREE ST(0) | C8 invalid | D0 FST ST(0) | D8 FSTP ST(0) | E0 FUCOM ST(0), ST(0) | E8 FUCOMP ST(0) | F0 invalid | F8 invalid |
| | | C1 FFREE ST(1) | C9 invalid | D1 FST ST(1) | D9 FSTP ST(1) | E1 FUCOM ST(1), ST(0) | E9 FUCOMP ST(1) | F1 invalid | F9 invalid |
| | | C2 FFREE ST(2) | CA invalid | D2 FST ST(2) | DA FSTP ST(2) | E2 FUCOM ST(2), ST(0) | EA FUCOMP ST(2) | F2 invalid | FA invalid |
| | | C3 FFREE ST(3) | CB invalid | D3 FST ST(3) | DB FSTP ST(3) | E3 FUCOM ST(3), ST(0) | EB FUCOMP ST(3) | F3 invalid | FB invalid |
| | | C4 FFREE ST(4) | CC invalid | D4 FST ST(4) | DC FSTP ST(4) | E4 FUCOM ST(4), ST(0) | EC FUCOMP ST(4) | F4 invalid | FC invalid |
| | | C5 FFREE ST(5) | CD invalid | D5 FST ST(5) | DD FSTP ST(5) | E5 FUCOM ST(5), ST(0) | ED FUCOMP ST(5) | F5 invalid | FD invalid |
| | | C6 FFREE ST(6) | CE invalid | D6 FST ST(6) | DE FSTP ST(6) | E6 FUCOM ST(6), ST(0) | EE FUCOMP ST(6) | F6 invalid | FE invalid |
| | | C7 FFREE ST(7) | CF invalid | D7 FST ST(7) | DF FSTP ST(7) | E7 FUCOM ST(7), ST(0) | EF FUCOMP ST(7) | F7 invalid | FF invalid |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|-----------|------------------------------|------------------------------------|------------------------------------|----------------------|----------------------|-------------------------------------|------------------------------------|-------------------------------------|------------------------------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DE | !11 | 00-BF | | | | | | | |
| | | FIADD mem16int | FIMUL mem16int | FICOM mem16int | FICOMP mem16int | FISUB mem16int | FISUBR mem16int | FIDIV mem16int | FIDIVR mem16int |
| | 11 | C0 FADDP ST(0), ST(0) | C8 FMULP ST(0), ST(0) | D0 invalid | D8 invalid | E0 FSUBRP ST(0), ST(0) | E8 FSUBP ST(0), ST(0) | F0 FDIVRP ST(0), ST(0) | F8 FDIVP ST(0), ST(0) |
| | | C1 FADDP ST(1), ST(0) | C9 FMULP ST(1), ST(0) | D1 invalid | D9 FCOMPP | E1 FSUBRP ST(1), ST(0) | E9 FSUBP ST(1), ST(0) | F1 FDIVRP ST(1), ST(0) | F9 FDIVP ST(1), ST(0) |
| | | C2 FADDP ST(2), ST(0) | CA FMULP ST(2), ST(0) | D2 invalid | DA invalid | E2 FSUBRP ST(2), ST(0) | EA FSUBP ST(2), ST(0) | F2 FDIVRP ST(2), ST(0) | FA FDIVP ST(2), ST(0) |
| | | C3 FADDP ST(3), ST(0) | CB FMULP ST(3), ST(0) | D3 invalid | DB invalid | E3 FSUBRP ST(3), ST(0) | EB FSUBP ST(3), ST(0) | F3 FDIVRP ST(3), ST(0) | FB FDIVP ST(3), ST(0) |
| | | C4 FADDP ST(4), ST(0) | CC FMULP ST(4), ST(0) | D4 invalid | DC invalid | E4 FSUBRP ST(4), ST(0) | EC FSUBP ST(4), ST(0) | F4 FDIVRP ST(4), ST(0) | FC FDIVP ST(4), ST(0) |
| | | C5 FADDP ST(5), ST(0) | CD FMULP ST(5), ST(0) | D5 invalid | DD invalid | E5 FSUBRP ST(5), ST(0) | ED FSUBP ST(5), ST(0) | F5 FDIVRP ST(5), ST(0) | FD FDIVP ST(5), ST(0) |
| | | C6 FADDP ST(6), ST(0) | CE FMULP ST(6), ST(0) | D6 invalid | DE invalid | E6 FSUBRP ST(6), ST(0) | EE FSUBP ST(6), ST(0) | F6 FDIVRP ST(6), ST(0) | FE FDIVP ST(6), ST(0) |
| | | C7 FADDP ST(7), ST(0) | CF FMULP ST(7), ST(0) | D7 invalid | DF invalid | E7 FSUBRP ST(7), ST(0) | EF FSUBP ST(7), ST(0) | F7 FDIVRP ST(7), ST(0) | FF FDIVP ST(7), ST(0) |

Table 4-10. x87 Opcodes and ModRM Extensions (continued)

| Opcode | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | | | | | | | |
|--------|------------------------------|------------------------|---------------|------------------|-------------------|-------------------|-------------------------------|------------------------------|-----------------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| DF | !11 | 00-BF | | | | | | | |
| | | FILD mem16int | invalid | FIST mem16int | FISTP mem16int | FBLD mem80dec | FILD m64int | FBSTP mem80dec | FISTP m64int |
| | 11 | C0 invalid | C8 invalid | D0 invalid | D8 invalid | E0 FSTSW AX | E8 FUCOMIP ST(0), ST(0) | F0 FCOMIP ST(0), ST(0) | F8 invalid |
| | | C1 invalid | C9 invalid | D1 invalid | D9 invalid | E1 invalid | E9 FUCOMIP ST(0), ST(1) | F1 FCOMIP ST(0), ST(1) | F9 invalid |
| | | C2 invalid | CA invalid | D2 invalid | DA invalid | E2 invalid | EA FUCOMIP ST(0), ST(2) | F2 FCOMIP ST(0), ST(2) | FA invalid |
| | | C3 invalid | CB invalid | D3 invalid | DB invalid | E3 invalid | EB FUCOMIP ST(0), ST(3) | F3 FCOMIP ST(0), ST(3) | FB invalid |
| | | C4 invalid | CC invalid | D4 invalid | DC invalid | E4 invalid | EC FUCOMIP ST(0), ST(4) | F4 FCOMIP ST(0), ST(4) | FC invalid |
| | | C5 invalid | CD invalid | D5 invalid | DD invalid | E5 invalid | ED FUCOMIP ST(0), ST(5) | F5 FCOMIP ST(0), ST(5) | FD invalid |
| | | C6 invalid | CE invalid | D6 invalid | DE invalid | E6 invalid | EE FUCOMIP ST(0), ST(6) | F6 FCOMIP ST(0), ST(6) | FE invalid |
| | | C7 invalid | CF invalid | D7 invalid | DF invalid | E7 invalid | EF FUCOMIP ST(0), ST(7) | F7 FCOMIP ST(0), ST(7) | FF invalid |

A.2.8 rFLAGS Condition Codes for x87 Opcodes

Table 4-11 shows the rFLAGS condition codes specified by the opcode and ModRM bytes of the FCMOV_{cc} instructions.

Table 4-11. rFLAGS Condition Codes for FCMOV_{cc}

| Opcode (hex) | ModRM <i>mod</i> Field | ModRM <i>reg</i> Field | rFLAGS Value | cc Mnemonic | Condition |
|-----------------|------------------------------|------------------------------|------------------|-------------|--------------------|
| DA | 11 | 000 | CF = 1 | B | Below |
| | | 001 | ZF = 1 | E | Equal |
| | | 010 | CF = 1 or ZF = 1 | BE | Below or Equal |
| | | 011 | PF = 1 | U | Unordered |
| DB | | 000 | CF = 0 | NB | Not Below |
| | | 001 | ZF = 0 | NE | Not Equal |
| | | 010 | CF = 0 or ZF = 0 | NBE | Not Below or Equal |
| | | 011 | PF = 0 | NU | Not Unordered |

A.3 Operand Encodings

Register and memory operands are encoded using the *mode-register-memory* (ModRM) and the *scale-index-base* (SIB) bytes that follow the opcodes. In some instructions, the ModRM byte is followed by a SIB byte, which defines the instruction's memory-addressing mode for the complex-addressing modes.

A.3.1 ModRM Operand References

Figure 4-2 shows the format of a ModRM byte. There are three fields—*mod*, *reg*, and *r/m*. The *reg* field not only provides additional opcode bits—as described above beginning with “ModRM Extensions to One-Byte and Two-Byte Opcodes” on page 394 and ending with “x87 Encodings” on page 400—but is also used with the other two fields to specify operands. The *mod* and *r/m* fields are used together with each other and, in 64-bit mode, with the REX.R and REX.B bits of the REX prefix, to specify the location of the instruction's operands and certain of the possible addressing modes (specifically, the non-complex modes).

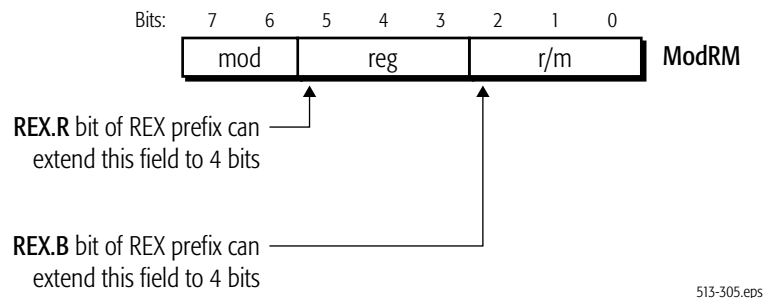


Figure 4-2. ModRM-Byte Format

The two sections below describe the ModRM operand encodings, first for 32-bit and 64-bit references, and then for 16-bit references.

32-Bit and 64-Bit Register and Memory References. Table 4-12 shows the encoding for 32-bit and 64-bit register references using the ModRM *reg* field. The first nine rows of Table 4-12 show references when the REX.R bit is cleared to 0, and the last nine rows show references when the REX.R bit is set to 1. In this table, *Mnemonic Notation* means the syntax notation shown in “Mnemonic Syntax” on page 40 for a register, and *ModRM Notation (/r)* means the opcode-syntax notation shown in “Opcode Syntax” on page 44 for the register.

Table 4-13 shows the encoding for 32-bit and 64-bit memory references using the ModRM byte. This table describes 32-bit and 64-bit addressing, with the REX.B bit set or cleared. The *Effective Address* is shown in the two left-most columns, followed by the binary encoding of the ModRM-byte *mod* field, followed by the eight possible hex values of the complete ModRM byte (one value for each binary encoding of the ModRM-byte *reg* field), followed by the binary encoding of the ModRM *r/m* field.

The /0 through /7 notation for the ModRM *reg* field (bits 5–3) means that the three-bit field contains a value from zero (binary 000) to 7 (binary 111).

Table 4-12. ModRM Register References, 32/64-Bit Address

| Mnemonic Notation | REX.R Bit | ModRM <i>reg</i> Field | | | | | | | |
|-------------------|-----------|------------------------|------|-------|-------|--------|--------|----------|----------|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| reg8 | 0 | AL | CL | DL | BL | AH/SPL | CH/BPL | DH/SIL | BH/DIL |
| reg16 | | AX | CX | DX | BX | SP | BP | SI | DI |
| reg32 | | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| reg64 | | RAX | RCX | RDX | RBX | RSP | RBP | RSI | RDI |
| mmx | | MMX0 | MMX1 | MMX2 | MMX3 | MMX4 | MMX5 | MMX6 | MMX7 |
| xmm | | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 |
| sReg | | ES | CS | SS | DS | FS | GS | reserved | reserved |
| cReg | | CR0 | CR1 | CR2 | CR3 | CR4 | CR5 | CR6 | CR7 |
| dReg | | DR0 | DR1 | DR2 | DR3 | DR4 | DR5 | DR6 | DR7 |
| reg8 | 1 | R8B | R9B | R10B | R11B | R12B | R13B | R14B | R15B |
| reg16 | | R8W | R9W | R10W | R11W | R12W | R13W | R14W | R15W |
| reg32 | | R8D | R9D | R10D | R11D | R12D | R13D | R14D | R15D |
| reg64 | | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
| mmx | | MMX0 | MMX1 | MMX2 | MMX3 | MMX4 | MMX5 | MMX6 | MMX7 |
| xmm | | XMM8 | XMM9 | XMM10 | XMM11 | XMM12 | XMM13 | XMM14 | XMM15 |
| sReg | | ES | CS | SS | DS | FS | GS | reserved | reserved |
| cReg | | CR8 | CR9 | CR10 | CR11 | CR12 | CR13 | CR14 | CR15 |
| dReg | | DR8 | DR9 | DR10 | DR11 | DR12 | DR13 | DR14 | DR15 |

Table 4-13. ModRM Memory References, 32/64-Bit Address

| Effective Address ¹ | | ModRM mod Field (binary) | ModRM reg Field ³ | | | | | | | | ModRM r/m Field (binary) |
|---------------------------------------|---------------------------------------|-----------------------------------|------------------------------|----|----|----|----|----|----|----|-----------------------------------|
| | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 | |
| REX.B = 0 | REX.B = 1 | | Complete ModRM Byte (hex) | | | | | | | | |
| [rAX] | [r8] | 00 | 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 | 000 |
| [rCX] | [r9] | | 01 | 09 | 11 | 19 | 21 | 29 | 31 | 39 | 001 |
| [rDX] | [r10] | | 02 | 0A | 12 | 1A | 22 | 2A | 32 | 3A | 010 |
| [rBX] | [r11] | | 03 | 0B | 13 | 1B | 23 | 2B | 33 | 3B | 011 |
| [sib] | [sib] | | 04 | 0C | 14 | 1C | 24 | 2C | 34 | 3C | 100 |
| [RIP+disp32] or [disp32] ² | [rIP+disp32] or [disp32] ² | | 05 | 0D | 15 | 1D | 25 | 2D | 35 | 3D | 101 |
| [rSI] | [r14] | | 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E | 110 |
| [rDI] | [r15] | | 07 | 0F | 17 | 1F | 27 | 2F | 37 | 3F | 111 |
| [rAX+disp8] | [r8+disp8] | 01 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 | 000 |
| [rCX+disp8] | [r9+disp8] | | 41 | 49 | 51 | 59 | 61 | 69 | 71 | 79 | 001 |
| [rDX+disp8] | [r10+disp8] | | 42 | 4A | 52 | 5A | 62 | 6A | 72 | 7A | 010 |
| [rBX+disp8] | [r11+disp8] | | 43 | 4B | 53 | 5B | 63 | 6B | 73 | 7B | 011 |
| [SIB-byte+disp8] | [SIB-byte+disp8] | | 44 | 4C | 54 | 5C | 64 | 6C | 74 | 7C | 100 |
| [rBP+disp8] | [r13+disp8] | | 45 | 4D | 55 | 5D | 65 | 6D | 75 | 7D | 101 |
| [rSI+disp8] | [r14+disp8] | | 46 | 4E | 56 | 5E | 66 | 6E | 76 | 7E | 110 |
| [rDI+disp8] | [r15+disp8] | | 47 | 4F | 57 | 5F | 67 | 6F | 77 | 7F | 111 |
| [rAX+disp32] | [r8+disp32] | 10 | 80 | 88 | 90 | 98 | A0 | A8 | B0 | B8 | 000 |
| [rCX+disp32] | [r9+disp32] | | 81 | 89 | 91 | 99 | A1 | A9 | B1 | B9 | 001 |
| [rDX+disp32] | [r10+disp32] | | 82 | 8A | 92 | 9A | A2 | AA | B2 | BA | 010 |
| [rBX+disp32] | [r11+disp32] | | 83 | 8B | 93 | 9B | A3 | AB | B3 | BB | 011 |
| [sib+disp32] | [sib+disp32] | | 84 | 8C | 94 | 9C | A4 | AC | B4 | BC | 100 |
| [rBP+disp32] | [r13+disp32] | | 85 | 8D | 95 | 9D | A5 | AD | B5 | BD | 101 |
| [rSI+disp32] | [r14+disp32] | | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE | 110 |
| [rDI+disp32] | [r15+disp32] | | 87 | 8F | 97 | 9F | A7 | AF | B7 | BF | 111 |

Note:

1. "disp8" and "disp32" indicate an 8-bit or 32-bit displacement.
2. In 64-bit mode, the effective address is [RIP+disp32]. In compatibility mode, the effective address is [disp32]. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP+disp32] effective address is truncated after computation to 32 bits.
3. See Table 4-12 for complete specification of ModRM "reg" field.

Table 4-13. ModRM Memory References, 32/64-Bit Address (continued)

| Effective Address ¹ | | ModRM mod Field (binary) | ModRM reg Field ³ | | | | | | | | ModRM r/m Field (binary) |
|---|----------------|-----------------------------------|------------------------------|----|----|----|----|----|----|----|-----------------------------------|
| | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 | |
| REX.B = 0 | REX.B = 1 | | Complete ModRM Byte (hex) | | | | | | | | |
| AL/rAX/MMX0/XMM0 | r8/MMX0/XMM8 | 11 | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 | 000 |
| CL/rCX/MMX1/XMM1 | r9/MMX1/XMM9 | | C1 | C9 | D1 | D9 | E1 | E9 | F1 | F9 | 001 |
| DL/rDX/MMX2/XMM2 | r10/MMX2/XMM10 | | C2 | CA | D2 | DA | E2 | EA | F2 | FA | 010 |
| BL/rBX/MMX3/XMM3 | r11/MMX3/XMM11 | | C3 | CB | D3 | DB | E3 | EB | F3 | FB | 011 |
| AH/SPL/rSP/MMX4/XMM4 | r12/MMX4/XMM12 | | C4 | CC | D4 | DC | E4 | EC | F4 | FC | 100 |
| CH/BPL/rBP/MMX5/XMM5 | r13/MMX5/XMM13 | | C5 | CD | D5 | DD | E5 | ED | F5 | FD | 101 |
| DH/SIL/rSI/MMX6/XMM6 | r14/MMX6/XMM14 | | C6 | CE | D6 | DE | E6 | EE | F6 | FE | 110 |
| BHDIL/rDI/MMX7/XMM7 | r15/MMX7/XMM15 | | C7 | CF | D7 | DF | E7 | EF | F7 | FF | 111 |
| Note: 1. “disp8” and “disp32” indicate an 8-bit or 32-bit displacement. 2. In 64-bit mode, the effective address is [RIP+disp32]. In compatibility mode, the effective address is [disp32]. If the address-size prefix is used in 64-bit mode to override 64-bit addressing, the [RIP+disp32] effective address is truncated after computation to 32 bits. 3. See Table 4-12 for complete specification of ModRM “reg” field. | | | | | | | | | | | |

16-Bit Register and Memory References. Table 4-14 shows the notation and encoding conventions for 16-bit register references using the ModRM reg field. This table is comparable to Table 4-12 but applies only to 16-bit operands (the 66h 16-bit operand-size prefix cannot be used with a REX operand-size prefix). Table 4-15 shows the notation and encoding conventions for 16-bit memory references using the ModRM byte. This table is comparable to Table 4-13.

Table 4-14. ModRM Register References, 16-Bit Address

| Mnemonic Notation | ModRM reg Field | | | | | | | |
|----------------------|-----------------|------|------|------|------|------|------|------|
| | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| reg8 | AL | CL | DL | BL | AH | CH | DH | BH |
| reg16 | AX | CX | DX | BX | SP | BP | SI | DI |
| reg32 | EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |
| mmx | MMX0 | MMX1 | MMX2 | MMX3 | MMX4 | MMX5 | MMX6 | MMX7 |
| xmm | XMM0 | XMM1 | XMM2 | XMM3 | XMM4 | XMM5 | XMM6 | XMM7 |

Table 4-14. ModRM Register References, 16-Bit Address (continued)

| Mnemonic Notation | ModRM <i>reg</i> Field | | | | | | | |
|-------------------|------------------------|-----|-----|-----|-----|-----|----------|----------|
| | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| sReg | ES | CS | SS | DS | FS | GS | reserved | reserved |
| cReg | CR0 | CR1 | CR2 | CR3 | CR4 | CR5 | CR6 | CR7 |
| dReg | DR0 | DR1 | DR2 | DR3 | DR4 | DR5 | DR6 | DR7 |

Table 4-15. ModRM Memory References, 16-Bit Address

| Effective Address ¹ | ModRM mod Field (binary) | ModRM <i>reg</i> Field ² | | | | | | | | ModRM <i>r/m</i> Field (binary) |
|--------------------------------|-----------------------------------|-------------------------------------|----|----|----|----|----|----|----|--|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 | |
| | | Complete ModRM Byte (hex) | | | | | | | | |
| [BX+SI] | 00 | 00 | 08 | 10 | 18 | 20 | 28 | 30 | 38 | 000 |
| [BX+DI] | | 01 | 09 | 11 | 19 | 21 | 29 | 31 | 39 | 001 |
| [BP+SI] | | 02 | 0A | 12 | 1A | 22 | 2A | 32 | 3A | 010 |
| [BP+DI] | | 03 | 0B | 13 | 1B | 23 | 2B | 33 | 3B | 011 |
| [SI] | | 04 | 0C | 14 | 1C | 24 | 2C | 34 | 3C | 100 |
| [DI] | | 05 | 0D | 15 | 1D | 25 | 2D | 35 | 3D | 101 |
| [disp16] | | 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E | 110 |
| [BX] | | 07 | 0F | 17 | 1F | 27 | 2F | 37 | 3F | 111 |
| [BX+SI+disp8] | 01 | 40 | 48 | 50 | 58 | 60 | 68 | 70 | 78 | 000 |
| [BX+DI+disp8] | | 41 | 49 | 51 | 59 | 61 | 69 | 71 | 79 | 001 |
| [BP+SI+disp8] | | 42 | 4A | 52 | 5A | 62 | 6A | 72 | 7A | 010 |
| [BP+DI+disp8] | | 43 | 4B | 53 | 5B | 63 | 6B | 73 | 7B | 011 |
| [SI+disp8] | | 44 | 4C | 54 | 5C | 64 | 6C | 74 | 7C | 100 |
| [DI+disp8] | | 45 | 4D | 55 | 5D | 65 | 6D | 75 | 7D | 101 |
| [BP+disp8] | | 46 | 4E | 56 | 5E | 66 | 6E | 76 | 7E | 110 |
| [BX+disp8] | | 47 | 4F | 57 | 5F | 67 | 6F | 77 | 7F | 111 |

Note:

1. “disp8” and “disp16” indicate an 8-bit or 16-bit displacement.
2. See Table 4-14 for complete specification of ModRM “reg” field.

Table 4-15. ModRM Memory References, 16-Bit Address (continued)

| Effective Address ¹ | ModRM mod Field (binary) | ModRM <i>reg</i> Field ² | | | | | | | | ModRM <i>r/m</i> Field (binary) |
|--|-----------------------------------|-------------------------------------|----|----|----|----|----|----|----|--|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 | |
| | | Complete ModRM Byte (hex) | | | | | | | | |
| [BX+SI+ <i>disp 16</i>] | 10 | 80 | 88 | 90 | 98 | A0 | A8 | B0 | B8 | 000 |
| [BX+DI+ <i>disp 16</i>] | | 81 | 89 | 91 | 99 | A1 | A9 | B1 | B9 | 001 |
| [BP+SI+ <i>disp 16</i>] | | 82 | 8A | 92 | 9A | A2 | AA | B2 | BA | 010 |
| [BP+DI+ <i>disp 16</i>] | | 83 | 8B | 93 | 9B | A3 | AB | B3 | BB | 011 |
| [SI+ <i>disp 16</i>] | | 84 | 8C | 94 | 9C | A4 | AC | B4 | BC | 100 |
| [DI+ <i>disp 16</i>] | | 85 | 8D | 95 | 9D | A5 | AD | B5 | BD | 101 |
| [BP+ <i>disp 16</i>] | | 86 | 8E | 96 | 9E | A6 | AE | B6 | BE | 110 |
| [BX+ <i>disp 16</i>] | | 87 | 8F | 97 | 9F | A7 | AF | B7 | BF | 111 |
| AL/AX/EAX/MMX0/XMM0 | 11 | C0 | C8 | D0 | D8 | E0 | E8 | F0 | F8 | 000 |
| CL/CX/ECX/MMX1/XMM1 | | C1 | C9 | D1 | D9 | E1 | E9 | F1 | F9 | 001 |
| DL/DX/EDX/MMX2/XMM2 | | C2 | CA | D2 | DA | E2 | EA | F2 | FA | 010 |
| BL/BX/EBX/MMX3/XMM3 | | C3 | CB | D3 | DB | E3 | EB | F3 | FB | 011 |
| AH/SP/ESP/MMX4/XMM4 | | C4 | CC | D4 | DC | E4 | EC | F4 | FC | 100 |
| CH/BP/EBP/MMX5/XMM5 | | C5 | CD | D5 | DD | E5 | ED | F5 | FD | 101 |
| DH/SI/ESI/MMX6/XMM6 | | C6 | CE | D6 | DE | E6 | EE | F6 | FE | 110 |
| BH/DI/EDI/MMX7/XMM7 | | C7 | CF | D7 | DF | E7 | EF | F7 | FF | 111 |
| Note: 1. “ <i>disp8</i> ” and “ <i>disp16</i> ” indicate an 8-bit or 16-bit displacement. 2. See Table 4-14 for complete specification of ModRM “ <i>reg</i> ” field. | | | | | | | | | | |

A.3.2 SIB Operand References

Figure 4-3 on page 416 shows the format of a scale-index-base (SIB) byte. Some instructions have a SIB byte following their ModRM byte to define memory addressing for the complex-addressing modes described in “Effective Addresses” in Volume 1. The SIB byte has three fields—*scale*, *index*, and *base*—that define the scale factor, index-register number, and base-register number for 32-bit and 64-bit complex addressing modes. In 64-bit mode, the REX.B and REX.X bits extend the encoding of the SIB byte’s *base* and *index* fields.

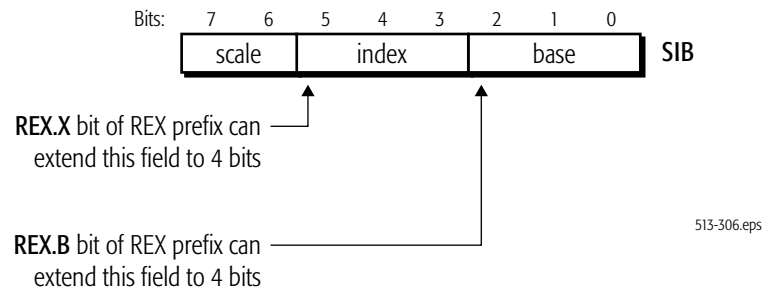
**Figure 4-3. SIB-Byte Format**

Table 4-16 shows the encodings for the SIB byte's *base* field, which specifies the base register for addressing. Table 4-17 on page 417 shows the encodings for the effective address referenced by a complete SIB byte, including its *scale* and *index* fields. The /0 through /7 notation for the SIB *base* field means that the three-bit field contains a value between zero (binary 000) and 7 (binary 111).

Table 4-16. SIB *base* Field References

| REX.B Bit | ModRM <i>mod</i> Field | SIB <i>base</i> Field | | | | | | | |
|-----------|------------------------|-----------------------|-----|-----|-----|-----|-------------------|-----|-----|
| | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| 0 | 00 | rAX | rCX | rDX | rBX | rSP | <i>disp32</i> | rSI | rDI |
| | 01 | | | | | | <i>rBP+disp8</i> | | |
| | 10 | | | | | | <i>rBP+disp32</i> | | |
| 1 | 00 | r8 | r9 | r10 | r11 | r12 | <i>disp32</i> | r14 | r15 |
| | 01 | | | | | | <i>r13+disp8</i> | | |
| | 10 | | | | | | <i>r13+disp32</i> | | |

Table 4-17. SIB Memory References

| Effective Address | | SIB scale Field | SIB index Field | SIB base Field ¹ | | | | | | | | |
|---|--------------|-----------------------|-----------------------|-----------------------------|-----|-----|-----|-----|-----|-------------------|-----|-----|
| | | | | REX.B = 0: | rAX | rCX | rDX | rBX | rSP | note ¹ | rSI | rDI |
| | | | | REX.B = 1: | r8 | r9 | r10 | r11 | r12 | note ¹ | r14 | r15 |
| | | | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| REX.X = 0 | REX.X = 1 | | | Complete SIB Byte (hex) | | | | | | | | |
| [rAX+base] | [r8+base] | 00 | 000 | | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| [rCX+base] | [r9+base] | | 001 | | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| [rDX+base] | [r10+base] | | 010 | | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| [rBX+base] | [r11+base] | | 011 | | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F |
| [base] | [r12+base] | | 100 | | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| [rBP+base] | [r13+base] | | 101 | | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
| [rSI+base] | [r14+base] | | 110 | | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| [rDI+base] | [r15+base] | | 111 | | 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
| [rAX*2+base] | [r8*2+base] | 01 | 000 | | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| [rCX*2+base] | [r9*2+base] | | 001 | | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
| [rDX*2+base] | [r10*2+base] | | 010 | | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| [rBX*2+base] | [r11*2+base] | | 011 | | 58 | 59 | 5A | 5B | 5C | 5D | 5E | 5F |
| [base] | [r12*2+base] | | 100 | | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| [rBP*2+base] | [r13*2+base] | | 101 | | 68 | 69 | 6A | 6B | 6C | 6D | 6E | 6F |
| [rSI*2+base] | [r14*2+base] | | 110 | | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
| [rDI*2+base] | [r15*2+base] | | 111 | | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F |
| [rAX*4+base] | [r8*4+base] | 10 | 000 | | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 |
| [rCX*4+base] | [r9*4+base] | | 001 | | 88 | 89 | 8A | 8B | 8C | 8D | 8E | 8F |
| [rDX*4+base] | [r10*4+base] | | 010 | | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 |
| [rBX*4+base] | [r11*4+base] | | 011 | | 98 | 99 | 9A | 9B | 9C | 9D | 9E | 9F |
| [base] | [r12*4+base] | | 100 | | A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 |
| [rBP*4+base] | [r13*4+base] | | 101 | | A8 | A9 | AA | AB | AC | AD | AE | AF |
| [rSI*4+base] | [r14*4+base] | | 110 | | B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 |
| [rDI*4+base] | [r15*4+base] | | 111 | | B8 | B9 | BA | BB | BC | BD | BE | BF |
| Note: | | | | | | | | | | | | |
| 1. See Table 4-16 on page 416 for complete specification of SIB “base” field. | | | | | | | | | | | | |

Table 4-17. SIB Memory References (continued)

| Effective Address | | SIB <i>scale</i> Field | SIB <i>index</i> Field | SIB <i>base</i> Field ¹ | | | | | | | | |
|---|--------------|------------------------------|------------------------------|------------------------------------|-----|-----|-----|-----|-----|-------------------|-----|-----|
| | | | | REX.B = 0: | rAX | rCX | rDX | rBX | rSP | note ¹ | rSI | rDI |
| | | | | REX.B = 1: | r8 | r9 | r10 | r11 | r12 | note ¹ | r14 | r15 |
| | | | | | /0 | /1 | /2 | /3 | /4 | /5 | /6 | /7 |
| REX.X = 0 | REX.X = 1 | | | Complete SIB Byte (hex) | | | | | | | | |
| [rAX*8+base] | [r8*8+base] | 11 | 000 | | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| [rCX*8+base] | [r9*8+base] | | 001 | | C8 | C9 | CA | CB | CC | CD | CE | CF |
| [rDX*8+base] | [r10*8+base] | | 010 | | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 |
| [rBX*8+base] | [r11*8+base] | | 011 | | D8 | D9 | DA | DB | DC | DD | DE | DF |
| [base] | [r12*8+base] | | 100 | | E0 | E1 | E2 | E3 | E4 | E5 | E6 | E7 |
| [rBP*8+base] | [r13*8+base] | | 101 | | E8 | E9 | EA | EB | EC | ED | EE | EF |
| [rSI*8+base] | [r14*8+base] | | 110 | | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
| [rDI*8+base] | [r15*8+base] | | 111 | | F8 | F9 | FA | FB | FC | FD | FE | FF |
| Note: | | | | | | | | | | | | |
| 1. See Table 4-16 on page 416 for complete specification of SIB “base” field. | | | | | | | | | | | | |

Appendix B General-Purpose Instructions in 64-Bit Mode

This appendix provides details of the general-purpose instructions in 64-bit mode and its differences from legacy and compatibility modes. The appendix covers only the general-purpose instructions (those described in *Chapter 3, “General-Purpose Instruction Reference”*). It does not cover the 128-bit media, 64-bit media, or x87 floating-point instructions because those instructions are not affected by 64-bit mode, other than in the access by such instructions to extended GPR and XMM registers when using a REX prefix.

B.1 General Rules for 64-Bit Mode

In 64-bit mode, the following general rules apply to instructions and their operands:

- **“Promoted to 64 Bit”**: If an instruction’s operand size (16-bit or 32-bit) in legacy and compatibility modes depends on the CS.D bit and the operand-size override prefix, then the operand-size choices in 64-bit mode are extended from 16-bit and 32-bit to include 64 bits (with a REX prefix), or the operand size is fixed at 64 bits. Such instructions are said to be “*Promoted to 64 bits*” in Table B-1. However, byte-operand opcodes of such instructions are not promoted.
- **Byte-Operand Opcodes Not Promoted**: As stated above in “Promoted to 64 Bit”, byte-operand opcodes of promoted instructions are not promoted. Those opcodes continue to operate only on bytes.
- **Fixed Operand Size**: If an instruction’s operand size is fixed in legacy mode (thus, independent of CS.D and prefix overrides), that operand size is usually fixed at the same size in 64-bit mode. For example, CUID operates on 32-bit operands, irrespective of attempts to override the operand size. (There are some exceptions, however, such as BSWAP.)
- **Default Operand Size**: The default operand size for most instructions is 32 bits, and a REX prefix must be used to change the operand size to 64 bits. However, two groups of instructions default to 64-bit operand size and do not need a REX prefix: (1) near branches and (2) all instructions,

except far branches, that implicitly reference the RSP. See Table B-5 on page 453 for a list of all instructions that default to 64-bit operand size.

- **Zero-Extension of 32-Bit Results:** Operations on 32-bit operands in 64-bit mode zero-extend the high 32 bits of 64-bit GPR destination registers.
- **No Extension of 8-Bit and 16-Bit Results:** Operations on 8-bit and 16-bit operands in 64-bit mode leave the high 56 or 48 bits, respectively, of 64-bit GPR destination registers unchanged.
- **Shift and Rotate Counts:** When the operand size is 64 bits, shifts and rotates use one additional bit (6 bits total) to specify shift-count or rotate-count, allowing 64-bit shifts and rotates.
- **Immediates:** The maximum size of immediate operands is 32 bits, except that 64-bit immediates can be MOVED into 64-bit GPRs. In 64-bit mode, when the operand size is 64 bits, immediates are sign-extended to 64 bits during use, but their actual size (for value representation) remains a maximum of 32 bits.
- **Displacements:** The maximum size of an address displacement is 32 bits. In 64-bit mode, displacements are sign-extended to 64 bits during use, but their actual size (for value representation) remains a maximum of 32 bits.
- **Undefined High 32 Bits After Mode Change:** The processor does not preserve the upper 32 bits of the 64-bit GPRs across switches from 64-bit mode to compatibility or legacy modes. In compatibility or legacy mode, the upper 32 bits of the GPRs are undefined and not accessible to software.

B.2 Operation and Operand Size in 64-Bit Mode

Table B-1 lists the integer instructions, showing operand size in 64-bit mode and the state of the high 32 bits of destination registers when 32-bit operands are used. Opcodes, such as byte-operand versions of several instructions, that do not appear in Table B-1 are covered by the general rules described in “General Rules for 64-Bit Mode” on page 419.

Table B-1. Operations and Operands in 64-Bit Mode

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|---|--|---|
| AAA - ASCII Adjust after Addition | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 37 | | | | |
| AAD - ASCII Adjust AX before Division | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| D5 | | | | |
| AAM - ASCII Adjust AX after Multiply | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| D4 | | | | |
| AAS - ASCII Adjust AL after Subtraction | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 3F | | | | |
| ADC —Add with Carry | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 11 | | | | |
| 13 | | | | |
| 15 | | | | |
| 81 /2 | | | | |
| 83 /2 | | | | |
| Note: <div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div> <div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div> <div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div> <div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div> <div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div> <div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|---|-----------------------------------|--|--------------------------------------|
| ADD —Signed or Unsigned Add | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 01 | | | | |
| 03 | | | | |
| 05 | | | | |
| 81 /0 | | | | |
| 83 /0 | | | | |
| AND —Logical AND | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 21 | | | | |
| 23 | | | | |
| 25 | | | | |
| 81 /4 | | | | |
| 83 /4 | | | | |
| ARPL - Adjust Requestor Privilege Level | OPCODE USED as MOVSD in 64-BIT MODE | | | |
| 63 | | | | |
| BOUND - Check Array Against Bounds | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 62 | | | | |
| BSF —Bit Scan Forward | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F BC | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|--|--------------------------------------|
| BSR —Bit Scan Reverse | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F BD | | | | |
| BSWAP —Byte Swap | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F C8 | | | | |
| BT —Bit Test | Promoted to 64 bits. | 32 bits | No GPR register results. | |
| 0F A3 | | | | |
| 0F BA /4 | | | | |
| BTC —Bit Test and Complement | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F BA /7 | | | | |
| 0F BB | | | | |
| BTR —Bit Test and Reset | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F B3 | | | | |
| 0F BA /6 | | | | |
| BTS —Bit Test and Set | Promoted to 64 bits. | 32 bits | No GPR register results. | |
| 0F AB | | | | |
| 0F BA /5 | | | | |

Note:

- The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
- If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
- Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
- The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
- Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
- See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--|
| CALL —Procedure Call Near | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| E8 | Promoted to 64 bits. | 64 bits | RIP = RIP + 32-bit displacement sign-extended to 64 bits. (16-bit displacement size cannot be encoded.) | |
| FF /2 | Promoted to 64 bits. | 64 bits | RIP = 64-bit offset from register or memory. | |
| CALL —Procedure Call Far | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| 9A | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| FF /3 | Promoted to 64 bits. | 32 bits | If selector points to a gate, then RIP = zero-extended 32-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction. | If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction. |
| Note: <div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div> <div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div> <div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div> <div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div> <div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div> <div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|--|---|--|
| CBW, CWDE, CDQE —Convert Byte to Word, Convert Word to Doubleword, Convert Doubleword to Quadword | Promoted to 64 bits. | 32 bits (size of destination register) | CWDE: Converts word to doubleword. Zero-extends EAX to RAX. | CDQE (new mnemonic): Converts doubleword to quadword. RAX = sign-extended EAX. |
| 98 | | | | |
| CDQ | see CWD, CDQ, CQO | | | |
| CDQE (new mnemonic) | see CBW, CWDE, CDQE | | | |
| CDWE | see CBW, CWDE, CDQE | | | |
| CLC —Clear Carry Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| F8 | | | | |
| CLD —Clear Direction Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| FC | | | | |
| CLFLUSH —Cache Line Invalidate | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F AE /7 | | | | |
| CLI —Clear Interrupt Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| FA | | | | |
| CLTS —Clear Task-Switched Flag in CR0 | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F 06 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|-----------------------------------|--|---|
| CMC —Complement Carry Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| F5 | | | | |
| CMOVcc —Conditional Move | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. This occurs even if the condition is false. | |
| 0F 42 through 0F 47 | | | | |
| 0F 4C through 0F 4F | | | | |
| CMP —Compare | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 39 | | | | |
| 3B | | | | |
| 3D | | | | |
| 81 /7 | | | | |
| 83 /7 | | | | |
| CMPS, CMPSB, CMPSW, CMPSD, CMPSQ —Compare Strings | Promoted to 64 bits. | 32 bits | CMPSD: Compare String Doublewords. See footnote ⁵ | CMPSQ (new mnemonic): Compare String Quadwords See footnote ⁵ |
| A7 | | | | |
| CMPXCHG —Compare and Exchange | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F B1 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|---|--|---|
| CMPXCHG8B —Compare and Exchange Eight Bytes | Same as legacy mode. | Operand size fixed at 64 bits. | Zero-extends EDX and EAX to 64 bits. | |
| 0F C7 /1 | | | | |
| CPUID —Processor Identification | Same as legacy mode. | Operand size fixed at 32 bits. | Zero-extends 32-bit register results to 64 bits. | |
| 0F A2 | | | | |
| CQO (new mnemonic) | see CWD, CDQ, CQO | | | |
| CWD, CDQ, CQO —Convert Word to Doubleword, Convert Doubleword to Quadword, Convert Quadword to Double Quadword | Promoted to 64 bits. | 32 bits (size of destination register) | CDQ: Converts doubleword to quadword. Sign-extends EAX to EDX. Zero-extends EDX to RDX. EAX is unchanged. | CQO (new mnemonic): Converts quadword to double quadword. Sign-extends RAX to RDX. RAX is unchanged. |
| 99 | | | | |
| DAA - Decimal Adjust AL after Addition | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 27 | | | | |
| DAS - Decimal Adjust AL after Subtraction | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 2F | | | | |
| Note: <div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div> <div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div> <div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div> <div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div> <div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div> <div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--|-----------------------------------|--|---|
| DEC —Decrement by 1 | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| FF /1 | | | | |
| 48 through 4F | OPCODE USED as REX PREFIX in 64-BIT MODE | | | |
| DIV —Unsigned Divide | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX). |
| F7 /6 | | | | |
| ENTER —Create Procedure Stack Frame | Promoted to 64 bits. | 64 bits | Can't encode ⁴ | |
| C8 | | | | |
| HLT —Halt | Same as legacy mode. | Not relevant. | No GPR register results. | |
| F4 | | | | |
| IDIV —Signed Divide | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | RDX:RAX contain a 64-bit quotient (RAX) and 64-bit remainder (RDX). |
| F7 /7 | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--|-----------------------------------|--|--|
| IMUL - Signed Multiply | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| F7 /5 | | | | RDX:RAX = RAX * reg/mem64 (i.e., 128-bit result) |
| 0F AF | | | | reg64 = reg64 * reg/mem64 |
| 69 | | | | reg64 = reg/mem64 * imm32 |
| 6B | | | | reg64 = reg/mem64 * imm8 |
| IN —Input From Port | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| E5 | | | | |
| ED | | | | |
| INC —Increment by 1 | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| FF /0 | | | | |
| 40 through 47 | OPCODE USED as REX PREFIX in 64-BIT MODE | | | |
| Note: <div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div> <div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div> <div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div> <div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div> <div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div> <div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|---|
| INS, INSB, INSD, INSW —Input String | Same as legacy mode. | 32 bits | INSD: Input String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote ⁵ | |
| 6D | | | | |
| INT n —Interrupt to Vector | Promoted to 64 bits. | Not relevant. | See “Long-Mode Interrupt Control Transfers” in Volume 2. | |
| CD | | | | |
| INT3 —Interrupt to Debug Vector | | | | |
| CC | | | | |
| INTO - Interrupt to Overflow Vector | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| CE | | | | |
| INVD —Invalidate Internal Caches | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F 08 | | | | |
| INVLPB —Invalidate TLB Entry | Promoted to 64 bits. | Not relevant. | No GPR register results. | |
| 0F 01 /7 | | | | |
| IRET, IRETD, IRETQ —Interrupt Return | Promoted to 64 bits. | 32 bits | IRETD: Interrupt Return Doubleword. See “Long-Mode Interrupt Control Transfers” in Volume 2. | IRETQ (new mnemonic): Interrupt Return Quadword. See “Long-Mode Interrupt Control Transfers” in Volume 2. |
| CF | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|---|--------------------------------------|
| Jcc—Jump Conditional | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| 70 through 7F | Promoted to 64 bits. | 64 bits | RIP = RIP + 8-bit displacement sign-extended to 64 bits. | |
| 0F 80 through 0F 8F | Promoted to 64 bits. | 64 bits | RIP = RIP + 32-bit displacement sign-extended to 64 bits. (16-bit displacement size cannot be encoded.) | |
| JCXZ, JECXZ, JRCXZ—Jump on CX/ECX/RCX Zero | Promoted to 64 bits. | 64 bits | RIP = RIP + 8-bit displacement sign-extended to 64 bits. | |
| E3 | | | See footnote ⁵ | |
| JMP—Jump Near | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| EB | Promoted to 64 bits. | 64 bits | RIP = RIP + 8-bit displacement sign-extended to 64 bits. | |
| E9 | | | RIP = RIP + 32-bit displacement sign-extended to 64 bits. (16-bit displacement size cannot be encoded.) | |
| FF /4 | | | RIP = 64-bit offset from register or memory. | |
| JMP—Jump Far | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| EA | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| Note: | | | | |
| 1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics. | | | | |
| 2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored. | | | | |
| 3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits. | | | | |
| 4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode. | | | | |
| 5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits. | | | | |
| 6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table. | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--|
| FF /5 | Promoted to 64 bits. | 32 bits | If selector points to a gate, then RIP = zero-extended 32-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction. | If selector points to a gate, then RIP = 64-bit offset from gate, else RIP = zero-extended 32-bit offset from far pointer referenced in instruction. |
| LAHF - Load Status Flags into AH Register | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 9F | | | | |
| LAR —Load Access Rights Byte | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F 02 | | | | |
| LDS - Load DS Far Pointer | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| C5 | | | | |
| LEA —Load Effective Address | Promoted to 64 bits. | 32 | Zero-extends 32-bit register results to 64 bits. | |
| 8D | | | | |
| LEAVE —Delete Procedure Stack Frame | Promoted to 64 bits. | 64 bits | Can't encode ⁴ | |
| C9 | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--------------------------------------|
| LES - Load ES Far Pointer | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| C4 | | | | |
| LFENCE —Load Fence | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F AE /5 | | | | |
| LFS —Load FS Far Pointer | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F B4 | | | | |
| LGDT —Load Global Descriptor Table Register | Promoted to 64 bits. | Operand size fixed at 8+2 bytes. | No GPR register results. Loads 8-byte base and 2-byte limit. | |
| 0F 01 /2 | | | | |
| LGS —Load GS Far Pointer | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F B5 | | | | |
| LIDT —Load Interrupt Descriptor Table Register | Promoted to 64 bits. | Operand size fixed at 8+2 bytes. | No GPR register results. Loads 8-byte base and 2-byte limit. | |
| 0F 01 /3 | | | | |
| LLDT —Load Local Descriptor Table Register | Promoted to 64 bits. | Operand size fixed at 16 bits. | No GPR register results. References 64-bit-mode descriptor to load 64-bit base. | |
| 0F 00 /2 | | | | |
| LMSW —Load Machine Status Word | Same as legacy mode. | Operand size fixed at 16 bits. | No GPR register results. | |
| 0F 01 /6 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|-----------------------------------|--|--|
| LODS, LODSB, LODSW, LODSD, LODSQ —Load String | Promoted to 64 bits. | 32 bits | LODSD: Load String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote ⁵ | LODSQ (new mnemonic): Load String Quadwords. See footnote ⁵ |
| AD | | | | |
| LOOP —Loop | Promoted to 64 bits. | 64 bits | RIP = RIP + 8-bit displacement sign-extended to 64 bits. See footnote ⁵ | |
| E2 | | | | |
| LOOPZ, LOOPE —Loop if Zero/Equal | | | | |
| E1 | | | | |
| LOOPNZ, LOOPNE —Loop if Not Zero/Equal | | | | |
| E0 | | | | |
| LSL —Load Segment Limit | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F 03 | | | | |
| LSS —Load SS Segment Register | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F B2 | | | | |
| LTR —Load Task Register | Promoted to 64 bits. | Operand size fixed at 16 bits. | No GPR register results. References 64-bit-mode descriptor to load 64-bit base. | |
| 0F 00 /3 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|---|--|
| MFENCE —Memory Fence | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F AE /6 | | | | |
| MOV —Move | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 89 | | | | |
| 8B | | | | |
| C7 | | | | 32-bit immediate is sign-extended to 64 bits. |
| B8 through BF | | | 64-bit immediate. | |
| A1 (moffset) | | | Zero-extends 32-bit register results to 64 bits. Memory offsets are address-sized and default to 64 bits. | Memory offsets are address-sized and default to 64 bits. |
| A3 (moffset) | | | | |
| MOV —Move to/from Segment Registers | Same as legacy mode. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 8C | | | | |
| 8E | | | | |
| Note: <div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div> <div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div> <div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div> <div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div> <div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div> <div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|---|---|
| MOV(CR<i>n</i>) —Move to/from Control Registers | Promoted to 64 bits. | Operand size fixed at 64 bits. | The high 32 bits of control registers differ in their writability and reserved status. See “System Resources” in Volume 2 for details. | |
| 0F 22 | | | | |
| 0F 20 | | | | |
| MOV(DR<i>n</i>) —Move to/from Debug Registers | Promoted to 64 bits. | Operand size fixed at 64 bits. | The high 32 bits of debug registers differ in their writability and reserved status. See “Debug and Performance Resources” in Volume 2 for details. | |
| 0F 21 | | | | |
| 0F 23 | | | | |
| MOVD —Move Doubleword or Quadword | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F 6E | | | Zero-extends 32-bit register results to 128 bits. | Zero-extends 64-bit register results to 128 bits. |
| 0F 7E | | | | |
| 66 0F 6E | | | | |
| 66 0F 7E | | | | |
| MOVNTI —Move Non-Temporal Doubleword | Promoted to 64 bits. | 32 bits | No GPR register results. | |
| 0F C3 | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--|-----------------------------------|--|--|
| MOVS, MOVSB, MOVSW, MOVSD, MOVSQ —Move String | Promoted to 64 bits. | 32 bits | MOVSD: Move String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote ⁵ | MOVSQ (new mnemonic): Move String Quadwords. See footnote ⁵ |
| A5 | | | | |
| MOVSX —Move with Sign-Extend | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Moves byte to quadword. |
| 0F BE | | | | Moves word to quadword. |
| 0F BF | | | | |
| MOVSXD —Move with Sign-Extend Doubleword | New instruction, available only in 64-bit mode. (In other modes, this opcode is ARPL instruction.) | 32 bits | Zero-extends 32-bit register results to 64 bits. | Sign-extends 32 bit register results to 64 bits. |
| 63 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|--|---|
| MOVZX —Move with Zero-Extend | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Moves byte to quadword. |
| 0F B6 | | | | Moves word to quadword. |
| 0F B7 | | | | |
| MUL —Multiply Unsigned | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | RDX:RAX=RAX * quadword in register or memory. |
| F7 /4 | | | | |
| NEG —Negate Two's Complement | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| F7 /3 | | | | |
| NOP —No Operation | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 90 | | | | |
| NOT —Negate One's Complement | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| F7 /2 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See "General Rules for 64-Bit Mode" on page 419 for definitions of "Promoted to 64 bits" and related topics.
2. If "Type of Operation" is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See "General Rules for 64-Bit Mode" on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|--|--------------------------------------|
| OR —Logical OR | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 09 | | | | |
| 0B | | | | |
| 0D | | | | |
| 81 /1 | | | | |
| 83 /1 | | | | |
| OUT —Output to Port | Same as legacy mode. | 32 bits | No GPR register results. | |
| E7 | | | | |
| EF | | | | |
| OUTS, OUTSB, OUTSD, OUTSW —Output String | Same as legacy mode. | 32 bits | OUTSD: Output String Doublewords. No GPR register results. See footnote ⁵ | |
| 6F | | | | |
| POP —Pop Stack | Promoted to 64 bits. | 64 bits | Cannot encode ⁴ | |
| 8F /0 | | | | |
| 58 through 5F | | | | |
| POP —Pop (segment register from) Stack | Same as legacy mode. | 64 bits | Cannot encode ⁴ | |
| 0F A1 (POP FS) | | | | |
| 0F A9 (POP GS) | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ | |
|---|---|-----------------------------------|--------------------------------------|---|--|
| 1F (POP DS) | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | | |
| 07 (POP ES) | | | | | |
| 17 (POP SS) | | | | | |
| POPA, POPAD - Pop All to GPR Words or Doublewords | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | | |
| 61 | | | | | |
| POPF, POPFD, POPFQ —Pop to rFLAGS Word, Doublword, or Quadword | Promoted to 64 bits. | 64 bits | Cannot encode ⁴ | POPFQ (new mnemonic): Pops 64 bits off stack, writes low 32 bits into EFLAGS and zero-extends the high 32 bits of RFLAGS. | |
| 9D | | | | | |
| PREFETCH —Prefetch L1 Data-Cache Line | Same as legacy mode. | Not relevant. | No GPR register results. | | |
| 0F 0D /0 | | | | | |
| PREFETCH/level —Prefetch Data to Cache Level <i>level</i> | Same as legacy mode. | Not relevant. | No GPR register results. | | |
| 0F 18 /0-3 | | | | | |
| PREFETCHW —Prefetch L1 Data-Cache Line for Write | Same as legacy mode. | Not relevant. | No GPR register results. | | |
| 0F 0D /1 | | | | | |
| Note: <div><div>1.</div><div>The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div></div> <div><div>2.</div><div>If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div></div> <div><div>3.</div><div>Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div></div> <div><div>4.</div><div>The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div></div> <div><div>5.</div><div>Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div></div> <div><div>6.</div><div>See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|---|-----------------------------------|--------------------------------------|---|
| PUSH —Push onto Stack | Promoted to 64 bits. | 64 bits | Cannot encode ⁴ | |
| FF /6 | | | | |
| 50 through 57 | | | | |
| 6A | | | | |
| 68 | | | | |
| PUSH —Push (segment register) onto Stack | Promoted to 64 bits. | 64 bits | Cannot encode ⁴ | |
| 0F A0 (PUSH FS) | | | | |
| 0F A8 (PUSH GS) | | | | |
| 0E (PUSH CS) | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 1E (PUSH DS) | | | | |
| 06 (PUSH ES) | | | | |
| 16 (PUSH SS) | | | | |
| PUSHA, PUSHAD - Push All to GPR Words or Doublewords | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 60 | | | | |
| PUSHF, PUSHFD, PUSHFQ —Push rFLAGS Word, Doubleword, or Quadword onto Stack | Promoted to 64 bits. | 64 bits | Cannot encode ⁴ | PUSHFQ (new mnemonic): Pushes the 64-bit RFLAGS register. |
| 9C | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|---|--------------------------------------|
| RCL —Rotate Through Carry Left | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /2 | | | | |
| D3 /2 | | | | |
| C1 /2 | | | | |
| RCR —Rotate Through Carry Right | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /3 | | | | |
| D3 /3 | | | | |
| C1 /3 | | | | |
| RDMSR —Read Model-Specific Register | Same as legacy mode. | Not relevant. | RDX[31:0] contains MSR[63:32], RAX[31:0] contains MSR[31:0]. Zero-extends 32-bit register results to 64 bits. | |
| 0F 32 | | | | |
| RDPMC —Read Performance-Monitoring Counters | Same as legacy mode. | Not relevant. | RDX[31:0] contains MSR[63:32], RAX[31:0] contains MSR[31:0]. Zero-extends 32-bit register results to 64 bits. | |
| 0F 33 | | | | |
| RDTS —Read Time-Stamp Counter | Same as legacy mode. | Not relevant. | RDX[31:0] contains MSR[63:32], RAX[31:0] contains MSR[31:0]. Zero-extends 32-bit register results to 64 bits. | |
| 0F 31 | | | | |
| REP INS —Repeat Input String | Same as legacy mode. | 32 bits | See footnote ⁵ | |
| F3 6D | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--------------------------------------|
| REP LODS —Repeat Load String | Promoted to 64 bits. | 32 bits | Zero-extends EAX, to 64 bits. See footnote ⁵ | See footnote ⁵ |
| F3 AD | | | | |
| REP MOVS —Repeat Move String | Promoted to 64 bits. | 32 bits | See footnote ⁵ | |
| F3 A5 | | | | |
| REP OUTS —Repeat Output String to Port | Same as legacy mode. | 32 bits | See footnote ⁵ | |
| F3 6F | | | | |
| REP STOS —Repeat Store String | Promoted to 64 bits. | 32 bits | See footnote ⁵ | |
| F3 AB | | | | |
| REPx CMPS —Repeat Compare String | Promoted to 64 bits. | 32 bits | See footnote ⁵ | |
| F3 A7 | | | | |
| REPx SCAS —Repeat Scan String | Promoted to 64 bits. | 32 bits | Zero-extends EAX, to 64 bits. See footnote ⁵ | |
| F3 AF | | | | |
| RET —Return from Call Near | See “Near Branches in 64-Bit Mode” in Volume 1. | | | |
| C2 | Promoted to 64 bits. | 64 bits | No GPR results. | |
| C3 | | | | |
| RET —Return from Call Far | Promoted to 64 bits. | 32 bits | See “Control Transfers” in Volume 1 and “Control-Transfer Privilege Checks” in Volume 2. | |
| CB | | | | |
| CA | | | | |
| Note: <ol style="list-style-type: none">The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table. | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--------------------------------------|
| ROL —Rotate Left | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /0 | | | | |
| D3 /0 | | | | |
| C1 /0 | | | | |
| ROR —Rotate Right | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /1 | | | | |
| D3 /1 | | | | |
| C1 /1 | | | | |
| RSM —Resume from System Management Mode | New SMM state-save area. | Not relevant. | See “System-Management Mode” in Volume 2. | |
| 0F AA | | | | |
| SAHF - Store AH into Flags | INVALID IN 64-BIT MODE (invalid-opcode exception) | | | |
| 9E | | | | |
| SAL —Shift Arithmetic Left | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /4 | | | | |
| D3 /4 | | | | |
| C1 /4 | | | | |
| Note: | | | | |
| 1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics. | | | | |
| 2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored. | | | | |
| 3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits. | | | | |
| 4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode. | | | | |
| 5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits. | | | | |
| 6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table. | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|-----------------------------------|--|---|
| SAR —Shift Arithmetic Right | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /7 | | | | |
| D3 /7 | | | | |
| C1 /7 | | | | |
| SBB —Subtract with Borrow | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 19 | | | | |
| 1B | | | | |
| 1D | | | | |
| 81 /3 | | | | |
| 83 /3 | | | | |
| SCAS, SCASB, SCASW, SCASD, SCASQ —Scan String | Promoted to 64 bits. | 32 bits | SCASD: Scan String Doublewords. Zero-extends 32-bit register results to 64 bits. See footnote ⁵ | SCASQ (new mnemonic): Scan String Quadwords. See footnote ⁵ |
| AF | | | | |
| SETcc —Byte Set if Condition | Same as legacy mode. | Operand size fixed at 8 bits. | | |
| 0F 90 through 0F 9F | | | | |
| SFENCE —Store Fence | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F AE /7 | | | | |
| Note: 1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics. 2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored. 3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits. 4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode. 5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits. 6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table. | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|-----------------------------------|--|--------------------------------------|
| SGDT —Store Global Descriptor Table Register | Promoted to 64 bits. | Operand size fixed at 8+2 bytes. | No GPR register results. Stores 8-byte base and 2-byte limit. | |
| 0F 01 /0 | | | | |
| SHL —Shift Left | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /4 | | | | |
| D3 /4 | | | | |
| C1 /4 | | | | |
| SHLD —Shift Left Double | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| 0F A4 | | | | |
| 0F A5 | | | | |
| SHR —Shift Right | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| D1 /5 | | | | |
| D3 /5 | | | | |
| C1 /5 | | | | |
| SHRD —Shift Right Double | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | Uses 6-bit count. |
| 0F AC | | | | |
| 0F AD | | | | |
| SIDT —Store Interrupt Descriptor Table Register | Promoted to 64 bits. | Operand size fixed at 8+2 bytes. | No GPR register results. Stores 8-byte base and 2-byte limit. | |
| 0F 01 /1 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|---|--|
| SLDT —Store Local Descriptor Table Register | Same as legacy mode. | 32 | Zero-extends 2-byte LDT selector to 64 bits. | |
| 0F 00 /0 | | | | |
| SMSW —Store Machine Status Word | Same as legacy mode. | 32 | Zero-extends 2-byte MSW to 64 bits. | |
| 0F 01 /4 | | | | |
| STC —Set Carry Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| F9 | | | | |
| STD —Set Direction Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| FD | | | | |
| STI - Set Interrupt Flag | Same as legacy mode. | Not relevant. | No GPR register results. | |
| FB | | | | |
| STOS, STOSB, STOSW, STOSD, STOSQ —Store String | Promoted to 64 bits. | 32 bits | STOSD: Store String Doublewords. See footnote ⁵ | STOSQ (new mnemonic): Store String Quadwords. See footnote ⁵ |
| AB | | | | |
| STR —Store Task Register | Same as legacy mode. | 32 | Zero-extends 2-byte TR selector to 64 bits. | |
| 0F 00 /1 | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|---|-----------------------------------|--|--------------------------------------|
| SUB —Subtract | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 29 | | | | |
| 2B | | | | |
| 2D | | | | |
| 81 /5 | | | | |
| 83 /5 | | | | |
| SWAPGS —Swap GS Register with KernelGSbase MSR | New instruction, available only in 64-bit mode. (In other modes, this opcode is invalid.) | Not relevant. | See “SWAPGS Instruction” in Volume 2. | |
| 0F 01 /7 | | | | |
| SYSCALL —Fast System Call | Promoted to 64 bits. | Not relevant. | See “SYSCALL and SYSRET Instructions” in Volume 2 for details. | |
| 0F 05 | | | | |
| SYSENTER —System Call | INVALID IN LONG MODE (invalid-opcode exception) | | | |
| 0F 34 | | | | |
| SYSEXIT —System Return | INVALID IN LONG MODE (invalid-opcode exception) | | | |
| 0F 35 | | | | |
| SYSRET —Fast System Return | Promoted to 64 bits. | 32 bits | See “SYSCALL and SYSRET Instructions” in Volume 2 for details. | |
| 0F 07 | | | | |
| Note: <div><div>1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.</div><div>2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.</div><div>3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.</div><div>4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.</div><div>5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.</div><div>6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.</div></div> | | | | |

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|--|--------------------------------|-----------------------------------|---|--------------------------------------|
| TEST —Test Bits | Promoted to 64 bits. | 32 bits | No GPR register results. | |
| 85 | | | | |
| A9 | | | | |
| F7 /0 | | | | |
| UD2 —Undefined Operation | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F 0B | | | | |
| VERR —Verify Segment for Reads | Same as legacy mode. | Operand size fixed at 16 bits | No GPR register results. | |
| 0F 00 /4 | | | | |
| VERW —Verify Segment for Writes | Same as legacy mode. | Operand size fixed at 16 bits | No GPR register results. | |
| 0F 00 /5 | | | | |
| WAIT —Wait for Interrupt | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 9B | | | | |
| WBINVD —Writeback and Invalidate All Caches | Same as legacy mode. | Not relevant. | No GPR register results. | |
| 0F 09 | | | | |
| WRMSR —Write to Model-Specific Register | Same as legacy mode. | Not relevant. | No GPR register results. MSR[63:32] = RDX[31:0] MSR[31:0] = RAX[31:0] | |
| 0F 30 | | | | |

Note:

1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics.
2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored.
3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits.
4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode.
5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits.
6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table.

Table B-1. Operations and Operands in 64-Bit Mode (continued)

| Instruction and Opcode (hex) ⁶ | Type of Operation ¹ | Default Operand Size ² | For 32-Bit Operand Size ³ | For 64-Bit Operand Size ³ |
|---|--------------------------------|-----------------------------------|--|--------------------------------------|
| XADD —Exchange and Add | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 0F C1 | | | | |
| XCHG —Exchange Register/Memory with Register | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 87 | | | | |
| 90 | | | | |
| XLAT, XLATB - Table Look-up Translation | Same as legacy mode. | Operand size fixed at 8 bits. | Writes AL, preserves RAX[63:8]. | |
| D7 | | | | |
| XOR —Logical Exclusive OR | Promoted to 64 bits. | 32 bits | Zero-extends 32-bit register results to 64 bits. | |
| 31 | | | | |
| 33 | | | | |
| 35 | | | | |
| 81 /6 | | | | |
| 83 /6 | | | | |
| Note: | | | | |
| 1. The type of operation, excluding considerations of operand size or extension of results. See “General Rules for 64-Bit Mode” on page 419 for definitions of “Promoted to 64 bits” and related topics. | | | | |
| 2. If “Type of Operation” is 64 bits, a REX prefix is needed for 64-bit operand size, unless the instruction size defaults to 64 bits. If the operand size is fixed, operand-size overrides are silently ignored. | | | | |
| 3. Special actions in 64-bit mode, in addition to legacy-mode actions. Zero or sign extensions apply only to result operands, not source operands. Unless otherwise stated, 8-bit and 16-bit results leave the high 56 or 48 bits, respectively, of 64-bit destination registers unchanged. Immediates and branch displacements are sign-extended to 64 bits. | | | | |
| 4. The default operand size can be overridden to 16 bits with 66h prefix, but there is no 32-bit operand-size override in 64-bit mode. | | | | |
| 5. Any pointer registers (rDI, rSI) or count registers (rCX) are address-sized and default to 64 bits. For 32-bit address size, any pointer and count registers are zero-extended to 64 bits. | | | | |
| 6. See “General Rules for 64-Bit Mode” on page 419, for opcodes that do not appear in this table. | | | | |

B.3 Invalid and Reassigned Instructions in 64-Bit Mode

Table B-2 lists instructions that are illegal in 64-bit mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

Table B-3 lists instructions that are reassigned to different functions in 64-bit mode. Attempted use of these instructions generates the reassigned function.

Table B-4 lists instructions that are illegal in long mode. Attempted use of these instructions generates an invalid-opcode exception (#UD).

Table B-2. Invalid Instructions in 64-Bit Mode

| Mnemonic | Opcode (hex) | Description |
|-------------|--------------|-------------------------------------|
| AAA | 37 | ASCII Adjust After Addition |
| AAD | D5 | ASCII Adjust Before Division |
| AAM | D4 | ASCII Adjust After Multiply |
| AAS | 3F | ASCII Adjust After Subtraction |
| BOUND | 62 | Check Array Bounds |
| CALL (far) | 9A | Procedure Call Far (far absolute) |
| DAA | 27 | Decimal Adjust after Addition |
| DAS | 2F | Decimal Adjust after Subtraction |
| INTO | CE | Interrupt to Overflow Vector |
| JMP (far) | EA | Jump Far (absolute) |
| LAHF | 9F | Load Status Flags into AH Register |
| LDS | C5 | Load DS Far Pointer |
| LES | C4 | Load ES Far Pointer |
| POP DS | 1F | Pop Stack into DS Segment |
| POP ES | 07 | Pop Stack into ES Segment |
| POP SS | 17 | Pop Stack into SS Segment |
| POPA, POPAD | 61 | Pop All to GPR Words or Doublewords |
| PUSH CS | 0E | Push CS Segment Selector onto Stack |
| PUSH DS | 1E | Push DS Segment Selector onto Stack |
| PUSH ES | 06 | Push ES Segment Selector onto Stack |
| PUSH SS | 16 | Push SS Segment Selector onto Stack |

Table B-2. Invalid Instructions in 64-Bit Mode (continued)

| Mnemonic | Opcode (hex) | Description |
|-------------------------------|--------------|--|
| PUSHA, PUSHAD | 60 | Push All to GPR Words or Doublewords |
| SAHF | 9E | Store AH into Flags |
| Redundant Grp1 (undocumented) | 82 | Redundant encoding of group1 Eb,Ib opcodes |
| SALC (undocumented) | D6 | Set AL According to CF |

Table B-3. Reassigned Instructions in 64-Bit Mode

| Mnemonic | Opcode (hex) | Description |
|-------------|--------------|---|
| ARPL | 63 | Opcode for MOVSLD instruction in 64-bit mode. In all other modes, this is the Adjust Requestor Privilege Level instruction opcode. |
| DEC and INC | 40-4F | Decrement by 1, Increment by 1 (see “Single-Byte INC and DEC Instructions in 64-Bit Mode” on page 454). Two-byte versions of DEC and INC are still valid. |

Table B-4. Invalid Instructions in Long Mode

| Mnemonic | Opcode (hex) | Description |
|----------|--------------|---------------|
| SYSENTER | 0F 34 | System Call |
| SYSEXIT | 0F 35 | System Return |

B.4 Instructions with 64-Bit Default Operand Size

In 64-bit mode, two groups of instructions default to 64-bit operand size without the need for a REX prefix:

- *Near branches*—CALL, Jcc, JMP, LOOP, and RET.
- *All instructions, except far branches, that implicitly reference the RSP*—CALL, ENTER, LEAVE, POP, PUSH, and RET (CALL and RET are in both groups of instructions).

Table B-5 lists these instructions.

Table B-5. Instructions Defaulting to 64-Bit Operand Size

| Mnemonic | Opcode (hex) | Implicitly Reference RSP | Description |
|--------------------|---------------|--------------------------|---|
| CALL | E8, FF /2 | yes | Call Procedure Near |
| ENTER | C8 | yes | Create Procedure Stack Frame |
| Jcc | many | no | Jump Conditional Near |
| JMP | E9, EB, FF /4 | no | Jump Near |
| LEAVE | C9 | yes | Delete Procedure Stack Frame |
| LOOP | E2 | no | Loop |
| LOOPcc | E0, E1 | no | Loop Conditional |
| POP reg/mem | 8F /0 | yes | Pop Stack (register or memory) |
| POP reg | 58-5F | yes | Pop Stack (register) |
| POP FS | 0F A1 | yes | Pop Stack into FS Segment Register |
| POP GS | 0F A9 | yes | Pop Stack into GS Segment Register |
| POPF, POPFD, POPFQ | 9D | yes | Pop to rFLAGS Word, Doubleword, or Quadword |
| PUSH imm8 | 6A | yes | Push onto Stack (sign-extended byte) |
| PUSH imm32 | 68 | yes | Push onto Stack (sign-extended doubleword) |
| PUSH reg/mem | FF /6 | yes | Push onto Stack (register or memory) |
| PUSH reg | 50-57 | yes | Push onto Stack (register) |
| PUSH FS | 0F A0 | yes | Push FS Segment Register onto Stack |

Table B-5. Instructions Defaulting to 64-Bit Operand Size (continued)

| Mnemonic | Opcode (hex) | Implicitly Reference RSP | Description |
|-----------------------|--------------|--------------------------|--|
| PUSH GS | 0F A8 | yes | Push GS Segment Register onto Stack |
| PUSHF, PUSHFD, PUSHFQ | 9C | yes | Push rFLAGS Word, Doubleword, or Quadword onto Stack |
| RET | C2, C3 | yes | Return From Call (near) |

The 64-bit default operand size can be overridden to 16 bits using the 66h operand-size override. However, it is not possible to override the operand size to 32 bits because there is no 32-bit operand-size override prefix for 64-bit mode. See “Operand-Size Override Prefix” on page 5 for details.

B.5 Single-Byte INC and DEC Instructions in 64-Bit Mode

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values, as described in “REX Prefixes” on page 14. Therefore, these single-byte opcodes for INC and DEC are not available in 64-bit mode, although they are available in legacy and compatibility modes. The functionality of these INC and DEC instructions is still available in 64-bit mode, however, using the ModRM forms of those instructions (opcodes FF/0 and FF/1).

B.6 NOP in 64-Bit Mode

Programs written for the legacy x86 architecture commonly use opcode 90h (the XCHG EAX, EAX instruction) as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this legacy NOP use. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats XCHG EAX, EAX as a true NOP, regardless of operand size or the presence of a REX prefix.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is

specified using a REX prefix byte, using the two byte form of XCHG to exchange a register with itself will not result in a no-operation because the default operation size is 32 bits in 64-bit mode.

B.7 Segment Override Prefixes in 64-Bit Mode

In 64-bit mode, the CS, DS, ES, SS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as true segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” in Volume 2 for details.

Appendix C Differences between Long Mode and Legacy Mode

Table C-1 summarizes the major differences between 64-bit mode and legacy protected mode. The third column indicates differences between 64-bit mode and legacy mode. The fourth column indicates whether that difference also applies to compatibility mode.

Table C-1. Differences Between Long Mode and Legacy Mode

| Type | Subject | 64-Bit Mode Difference | Applies To Compatibility Mode? |
|--------------------------------|-------------------------|---|--------------------------------|
| Application Programming | Addressing | RIP-relative addressing available | no |
| | Data and Address Sizes | Default data size is 32 bits | |
| | | REX Prefix toggles data size to 64 bits | |
| | | Default address size is 64 bits | |
| | | Address size prefix toggles address size to 32 bits | |
| | Instruction Differences | Various opcodes are invalid or changed (see Table B-4 on page 452) | |
| | | MOV reg,imm32 becomes MOV reg,imm64 (with REX operand size prefix) | |
| | | REX is always enabled | |
| | | Direct-offset forms of MOV to or from accumulator become 64-bit offsets | |
| | | MOVD extended to MOV 64 bits between MMX registers and long GPRs (with REX operand-size prefix) | |

Table C-1. Differences Between Long Mode and Legacy Mode (continued)

| Type | Subject | 64-Bit Mode Difference | Applies To Compatibility Mode? |
|---------------------------|--|---|--------------------------------|
| System Programming | x86 Modes | Real and virtual-8086 modes not supported | yes |
| | Task Switching | Task switching not supported | yes |
| | Addressing | 64-bit virtual addresses | yes |
| | | 4-level paging structures | |
| | | PAE must always be enabled | |
| | Segmentation | CS, DS, ES, SS segment bases are ignored | no |
| | | CS, DS, ES, FS, GS, SS segment limits are ignored | |
| | | CS, DS, ES, SS Segment prefixes are ignored | |
| | Exception and Interrupt Handling | All pushes are 8 bytes | yes |
| | | IDT entries are expanded to 16 bytes | |
| | | SS is not changed for stack switch | |
| | | SS:RSP is pushed unconditionally | |
| | Call Gates | All pushes are 8 bytes | yes |
| | | 16-bit call gates are illegal | |
| | | 32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes. | |
| | | SS is not changed for stack switch | |
| | System-Descriptor Registers | GDT, IDT, LDT, TR base registers expanded to 64 bits | yes |
| | System-Descriptor Table Entries and Pseudo-descriptors | LGDT and LIDT use expanded 10-byte pseudo-descriptors. | no |
| | | LLDT and LTR use expanded 16-byte table entries. | |

Appendix D Instruction Subsets and CPUID Feature Sets

Table D-1 is an alphabetical list of the x86-64 instruction set, including the instructions from all five of the instruction subsets that make up the entire x86-64 instruction-set architecture:

- Chapter 3, “General-Purpose Instruction Reference.”
- Chapter 4, “System Instruction Reference.”
- “128-Bit Media Instruction Reference” in Volume 4.
- “64-Bit Media Instruction Reference” in Volume 5.
- “x87 Floating-Point Instruction Reference” in Volume 5.

Several instructions belong to—and are described in—multiple instruction subsets. Table D-1 shows the minimum current privilege level (CPL) required to execute each instruction and the instruction subset(s) to which the instruction belongs. For each instruction subset, the CPUID feature set(s) that enables the instruction is shown.

D.1 Instruction Subsets

Figure D-1 shows the relationship between the five instruction subsets and the CPUID feature sets. Dashed-line polygons represent the instruction subsets. Circles represent the major CPUID feature sets that enable various classes of instructions. (There are a few additional CPUID feature sets, not shown, each of which apply to only a few instructions.)

The overlapping of the 128-bit and 64-bit media instruction subsets indicates that these subsets share some common mnemonics. However, these common mnemonics either have distinct opcodes for each subset or they take operands in both the MMX and XMM register sets.

The horizontal axis of Figure D-1 shows how the subsets and CPUID feature sets have evolved over time.

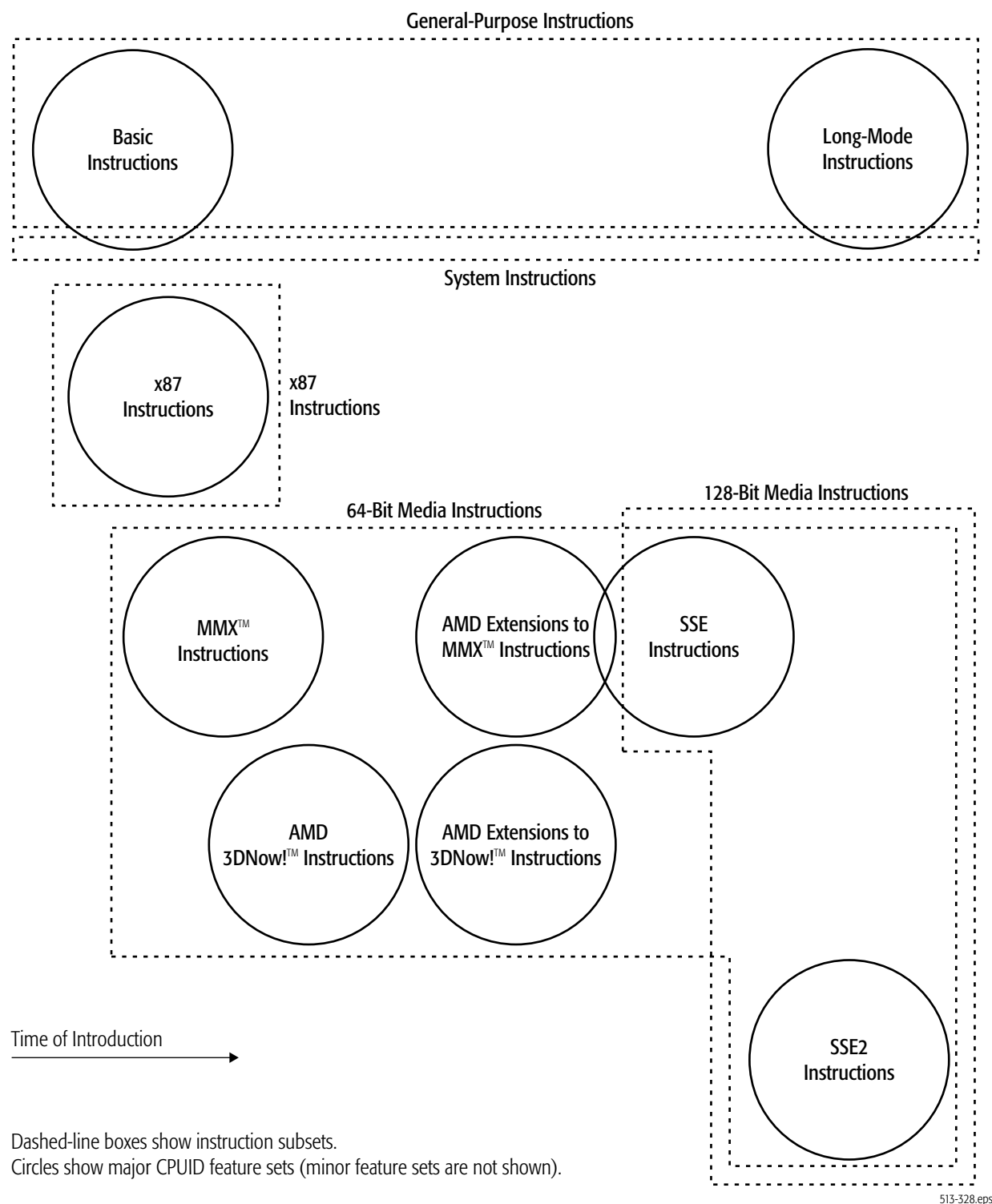


Figure D-1. Instruction Subsets vs. CPUID Feature Sets

D.2 CPUID Feature Sets

The CPUID feature sets shown in Figure D-1 and listed in Table D-1 on page 463 include:

- **Basic Instructions**—Instructions that are supported in all hardware implementations of the x86-64 architecture, except that the following instructions are implemented only if their associated CPUID function bit is set:
 - CMPXCHG8B, indicated by bit 8 of CPUID standard function 1 and extended function 8000_0001h.
 - CMOVcc (conditional moves), indicated by bit 15 of CPUID standard function 1 and extended function 8000_0001h.
 - RDMSR and WRMSR, indicated by bit 5 of CPUID standard function 1 and extended function 8000_0001h.
 - SYSCALL and SYSRET, indicated by bit 11 of CPUID extended function 8000_0001h.
 - SYSENTER and SYSEXIT, indicated by bit 11 of CPUID standard function 1.
- **x87 Instructions**—Legacy floating-point instructions that use the ST(0)–ST(7) stack registers (FPR0–FPR7 physical registers) and are supported if the following bits are set:
 - On-chip floating-point unit, indicated by bit 0 of CPUID standard function 1 and extended function 8000_0001h.
 - CMOVcc (conditional moves), indicated by bit 15 of CPUID standard function 1 and extended function 8000_0001h. This bit indicates support for x87 floating-point conditional moves (FCMOVcc) whenever the On-Chip Floating-Point Unit bit (bit 0) is also set.
- **MMX™ Instructions**—Vector integer instructions that are implemented in the MMX instruction set, use the MMX logical registers (FPR0–FPR7 physical registers), and are supported if the following bit is set:
 - MMX instructions, indicated by bit 23 of CPUID standard function 1 and extended function 8000_0001h.
- **AMD 3DNow!™ Instructions**—Vector floating-point instructions that comprise the AMD 3DNow! technology, use the MMX logical registers (FPR0–FPR7 physical registers), and are supported if the following bit is set:

- AMD 3DNow! instructions, indicated by bit 31 of CUID extended function 8000_0001h.
- *AMD Extensions to MMX™ Instructions*—Vector integer instructions that use the MMX registers and are supported if the following bit is set:
 - AMD extensions to MMX instructions, indicated by bit 22 of CUID extended function 8000_0001h.
- *AMD Extensions to 3DNow!™ Instructions*—Vector floating-point instructions that use the MMX registers and are supported if the following bit is set:
 - AMD extensions to 3DNow! instructions, indicated by bit 30 of CUID extended function 8000_0001h.
- *SSE Instructions*—Vector integer instructions that use the MMX registers, single-precision vector and scalar floating-point instructions that use the XMM registers, plus other instructions for data-type conversion, prefetching, cache control, and memory-access ordering. These instructions are supported if the following bits are set:
 - SSE, indicated by bit 25 of CUID extended function 8000_0001h.
 - FXSAVE and FXRSTOR, indicated by bit 24 of CUID standard function 1 and extended function 8000_0001h.

Several SSE opcodes are also implemented by the AMD Extensions to MMX™ Instructions.

- *SSE2 Instructions*—Vector and scalar integer and double-precision floating-point instructions that use the XMM registers, plus other instructions for data-type conversion, cache control, and memory-access ordering. These instructions are supported if the following bit is set:
 - SSE2, indicated by bit 26 of CUID extended function 8000_0001h.

Several instructions originally implemented as MMX™ instructions are extended in the SSE2 instruction set to include opcodes that use XMM registers.

- *Long-Mode Instructions*—Instructions introduced by AMD with the x86-64 architecture. These instructions are supported if the following bit is set:
 - Long mode, indicated by bit 29 of CUID extended function 8000_0001h.

For complete details on the CPUID feature sets listed in Table D-1, see “Processor Feature Identification” in Volume 2.

D.3 Instruction List

Table D-1. Instruction Subsets and CPUID Feature Sets

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| AAA | ASCII Adjust After Addition | 3 | Basic | | | | |
| AAD | ASCII Adjust Before Division | 3 | Basic | | | | |
| AAM | ASCII Adjust After Multiply | 3 | Basic | | | | |
| AAS | ASCII Adjust After Subtraction | 3 | Basic | | | | |
| ADC | Add with Carry | 3 | Basic | | | | |
| ADD | Signed or Unsigned Add | 3 | Basic | | | | |
| ADDPD | Add Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| ADDPS | Add Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| ADDSD | Add Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| ADDSS | Add Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| AND | Logical AND | 3 | Basic | | | | |
| ANDNPD | Logical Bitwise AND NOT Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| ANDNPS | Logical Bitwise AND NOT Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| ANDPD | Logical Bitwise AND Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| ANDPS | Logical Bitwise AND Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| ARPL | Adjust Requestor Privilege Level | 3 | | | | | Basic |
| BOUND | Check Array Bounds | 3 | Basic | | | | |
| BSF | Bit Scan Forward | 3 | Basic | | | | |
| BSR | Bit Scan Reverse | 3 | Basic | | | | |
| BSWAP | Byte Swap | 3 | Basic | | | | |
| BT | Bit Test | 3 | Basic | | | | |
| BTC | Bit Test and Complement | 3 | Basic | | | | |
| BTR | Bit Test and Reset | 3 | Basic | | | | |
| BTS | Bit Test and Set | 3 | Basic | | | | |
| CALL | Procedure Call | 3 | Basic | | | | |
| CBW | Convert Byte to Word | 3 | Basic | | | | |
| CDQ | Convert Doubleword to Quadword | 3 | Basic | | | | |
| CDQE | Convert Doubleword to Quadword | 3 | Long Mode | | | | |
| CLC | Clear Carry Flag | 3 | Basic | | | | |
| CLD | Clear Direction Flag | 3 | Basic | | | | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.

2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CLFLUSH | Cache Line Invalidate | 3 | SSE2 | | | | |
| CLI | Clear Interrupt Flag | 3 | | | | | Basic |
| CLTS | Clear Task-Switched Flag in CR0 | 0 | | | | | Basic |
| CMC | Complement Carry Flag | 3 | Basic | | | | |
| CMOVA | Conditional Move If Above | 3 | CMOVcc | | | | |
| CMOVAE | Conditional Move If Above or Equal | 3 | CMOVcc | | | | |
| CMOVB | Conditional Move If Below | 3 | CMOVcc | | | | |
| CMOVBE | Conditional Move If Below or Equal | 3 | CMOVcc | | | | |
| CMOVC | Conditional Move If Carry | 3 | CMOVcc | | | | |
| CMOVE | Conditional Move If Equal | 3 | CMOVcc | | | | |
| CMOVG | Conditional Move If Greater | 3 | CMOVcc | | | | |
| CMOVGE | Conditional Move If Greater or Equal | 3 | CMOVcc | | | | |
| CMOVL | Conditional Move If Less | 3 | CMOVcc | | | | |
| CMOVLE | Conditional Move If Less or Equal | 3 | CMOVcc | | | | |
| CMOVNA | Conditional Move If Not Above | 3 | CMOVcc | | | | |
| CMOVNAE | Conditional Move If Not Above or Equal | 3 | CMOVcc | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CMOVNB | Conditional Move If Not Below | 3 | CMOVcc | | | | |
| CMOVNBE | Conditional Move If Not Below or Equal | 3 | CMOVcc | | | | |
| CMOVNC | Conditional Move If Not Carry | 3 | CMOVcc | | | | |
| CMOVNE | Conditional Move If Not Equal | 3 | CMOVcc | | | | |
| CMOVNG | Conditional Move If Not Greater | 3 | CMOVcc | | | | |
| CMOVNGE | Conditional Move If Not Greater or Equal | 3 | CMOVcc | | | | |
| CMOVNL | Conditional Move If Not Less | 3 | CMOVcc | | | | |
| CMOVNLE | Conditional Move If Not Less or Equal | 3 | CMOVcc | | | | |
| CMOVNO | Conditional Move If No Overflow | 3 | CMOVcc | | | | |
| CMOVNP | Conditional Move If No Parity | 3 | CMOVcc | | | | |
| CMOVNS | Conditional Move If Not Sign | 3 | CMOVcc | | | | |
| CMOVNZ | Conditional Move If Not Zero | 3 | CMOVcc | | | | |
| CMOVO | Conditional Move If Overflow | 3 | CMOVcc | | | | |
| CMOVP | Conditional Move If Parity | 3 | CMOVcc | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|-------------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CMOVPE | Conditional Move If Parity Even | 3 | CMOVcc | | | | |
| CMOVPO | Conditional Move If Parity Odd | 3 | CMOVcc | | | | |
| CMOVS | Conditional Move If Sign | 3 | CMOVcc | | | | |
| CMOVZ | Conditional Move If Zero | 3 | CMOVcc | | | | |
| CMP | Compare | 3 | Basic | | | | |
| CMPPD | Compare Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| CMPPS | Compare Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| CMPS | Compare Strings | 3 | Basic | | | | |
| CMPSB | Compare Strings by Byte | 3 | Basic | | | | |
| CMPSD | Compare Strings by Doubleword | 3 | Basic ² | | | | |
| CMPSD | Compare Scalar Double-Precision Floating-Point | 3 | | SSE2 ² | | | |
| CMPSQ | Compare Strings by Quadword | 3 | Long Mode | | | | |
| CMPSD | Compare Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| CMPSW | Compare Strings by Word | 3 | Basic | | | | |
| CMPXCHG | Compare and Exchange | 3 | Basic | | | | |
| CMPXCHG8B | Compare and Exchange Eight Bytes | 3 | CMPXCHG8B | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CUID Feature Set(s) ¹ | | | | |
|--|--|-----|---|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| COMISD | Compare Ordered Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| COMISS | Compare Ordered Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| CPUID | Processor Identification | 3 | Basic | | | | |
| CQO | Convert Quadword to Double Quadword | 3 | Long Mode | | | | |
| CVTDQ2PD | Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| CVTDQ2PS | Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point | 3 | | SSE2 | | | |
| CVTPD2DQ | Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers | 3 | | SSE2 | | | |
| CVTPD2PI | Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers | 3 | | SSE2 | SSE2 | | |
| CVTPD2PS | Convert Packed Double-Precision Floating-Point to Packed Single-Precision Floating-Point | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CVTPI2PD | Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point | 3 | | SSE2 | SSE2 | | |
| CVTPI2PS | Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point | 3 | | SSE | SSE | | |
| CVTPS2DQ | Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers | 3 | | SSE2 | | | |
| CVTPS2PD | Convert Packed Single-Precision Floating-Point to Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| CVTPS2PI | Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers | 3 | | SSE | SSE | | |
| CVTSD2SI | Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer | 3 | | SSE2 | | | |
| CVTSD2SS | Convert Scalar Double-Precision Floating-Point to Scalar Single-Precision Floating-Point | 3 | | SSE2 | | | |
| CVTSI2SD | Convert Signed Doubleword or Quadword Integer to Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CVTSI2SS | Convert Signed Doubleword or Quadword Integer to Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| CVTSS2SD | Convert Scalar Single-Precision Floating-Point to Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| CVTSS2SI | Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer | 3 | | SSE | | | |
| CVTTPD2DQ | Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated | 3 | | SSE2 | | | |
| CVTTPD2PI | Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated | 3 | | SSE2 | SSE2 | | |
| CVTTPS2DQ | Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated | 3 | | SSE2 | | | |
| CVTTPS2PI | Convert Packed Single-Precision Floating-Point to Packed Doubleword Integers, Truncated | 3 | | SSE | SSE | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| CVTTSD2SI | Convert Scalar Double-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated | 3 | | SSE2 | | | |
| CVTTSS2SI | Convert Scalar Single-Precision Floating-Point to Signed Doubleword or Quadword Integer, Truncated | 3 | | SSE | | | |
| CWD | Convert Word to Doubleword | 3 | Basic | | | | |
| CWDE | Convert Word to Doubleword | 3 | Basic | | | | |
| DAA | Decimal Adjust after Addition | 3 | Basic | | | | |
| DAS | Decimal Adjust after Subtraction | 3 | Basic | | | | |
| DEC | Decrement by 1 | 3 | Basic | | | | |
| DIV | Unsigned Divide | 3 | Basic | | | | |
| DIVPD | Divide Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| DIVPS | Divide Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| DIVSD | Divide Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| DIVSS | Divide Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| EMMS | Enter/Exit Multimedia State | 3 | | | MMX | MMX | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.

2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|---|-----|--|---------------|--------------|-------------|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| ENTER | Create Procedure Stack Frame | 3 | Basic | | | | |
| F2XM1 | Floating-Point Compute $2x-1$ | 3 | | | | X87 | |
| FABS | Floating-Point Absolute Value | 3 | | | | X87 | |
| FADD | Floating-Point Add | 3 | | | | X87 | |
| FADDP | Floating-Point Add and Pop | 3 | | | | X87 | |
| FBLD | Floating-Point Load Binary-Coded Decimal | 3 | | | | X87 | |
| FBSTP | Floating-Point Store Binary-Coded Decimal Integer and Pop | 3 | | | | X87 | |
| FCHS | Floating-Point Change Sign | 3 | | | | X87 | |
| FCLEX | Floating-Point Clear Flags | 3 | | | | X87 | |
| FCMOVB | Floating-Point Conditional Move If Below | 3 | | | | X87, CMOVcc | |
| FCMOVBE | Floating-Point Conditional Move If Below or Equal | 3 | | | | X87, CMOVcc | |
| FCMOVE | Floating-Point Conditional Move If Equal | 3 | | | | X87, CMOVcc | |
| FCMOVNB | Floating-Point Conditional Move If Not Below | 3 | | | | X87, CMOVcc | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-------------|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FCMOVNBE | Floating-Point Conditional Move If Not Below or Equal | 3 | | | | X87, CMOVcc | |
| FCMOVNE | Floating-Point Conditional Move If Not Equal | 3 | | | | X87, CMOVcc | |
| FCMOVNU | Floating-Point Conditional Move If Not Unordered | 3 | | | | X87, CMOVcc | |
| FCMOVU | Floating-Point Conditional Move If Unordered | 3 | | | | X87, CMOVcc | |
| FCOM | Floating-Point Compare | 3 | | | | X87 | |
| FCOMI | Floating-Point Compare and Set Flags | 3 | | | | X87 | |
| FCOMIP | Floating-Point Compare and Set Flags and Pop | 3 | | | | X87 | |
| FCOMP | Floating-Point Compare and Pop | 3 | | | | X87 | |
| FCOMPP | Floating-Point Compare and Pop Twice | 3 | | | | X87 | |
| FCOS | Floating-Point Cosine | 3 | | | | X87 | |
| FDECSTP | Floating-Point Decrement Stack-Top Pointer | 3 | | | | X87 | |
| FDIV | Floating-Point Divide | 3 | | | | X87 | |
| FDIVP | Floating-Point Divide and Pop | 3 | | | | X87 | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|--------------|--------|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FDIVR | Floating-Point Divide Reverse | 3 | | | | X87 | |
| FDIVRP | Floating-Point Divide Reverse and Pop | 3 | | | | X87 | |
| FEMMS | Fast Enter/Exit Multimedia State | 3 | | | 3DNow! | 3DNow! | |
| FFREE | Free Floating-Point Register | 3 | | | | X87 | |
| FIADD | Floating-Point Add Integer to Stack Top | 3 | | | | X87 | |
| FICOM | Floating-Point Integer Compare | 3 | | | | X87 | |
| FICOMP | Floating-Point Integer Compare and Pop | 3 | | | | X87 | |
| FIDIV | Floating-Point Integer Divide | 3 | | | | X87 | |
| FIDIVR | Floating-Point Integer Divide Reverse | 3 | | | | X87 | |
| FILD | Floating-Point Load Integer | 3 | | | | X87 | |
| FIMUL | Floating-Point Integer Multiply | 3 | | | | X87 | |
| FINCSTP | Floating-Point Increment Stack-Top Pointer | 3 | | | | X87 | |
| FINIT | Floating-Point Initialize | 3 | | | | X87 | |
| FIST | Floating-Point Integer Store | 3 | | | | X87 | |
| FISTP | Floating-Point Integer Store and Pop | 3 | | | | X87 | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.

2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FISUB | Floating-Point Integer Subtract | 3 | | | | X87 | |
| FISUBR | Floating-Point Integer Subtract Reverse | 3 | | | | X87 | |
| FLD | Floating-Point Load | 3 | | | | X87 | |
| FLD1 | Floating-Point Load +1.0 | 3 | | | | X87 | |
| FLDCW | Floating-Point Load x87 Control Word | 3 | | | | X87 | |
| FLDENV | Floating-Point Load x87 Environment | 3 | | | | X87 | |
| FLDL2E | Floating-Point Load $\log_2 e$ | 3 | | | | X87 | |
| FLDL2T | Floating-Point Load $\log_2 10$ | 3 | | | | X87 | |
| FLDLG2 | Floating-Point Load $\log_{10} 2$ | 3 | | | | X87 | |
| FLDLN2 | Floating-Point Load Ln 2 | 3 | | | | X87 | |
| FLDPI | Floating-Point Load Pi | 3 | | | | X87 | |
| FLDZ | Floating-Point Load +0.0 | 3 | | | | X87 | |
| FMUL | Floating-Point Multiply | 3 | | | | X87 | |
| FMULP | Floating-Point Multiply and Pop | 3 | | | | X87 | |
| FNCLEX | Floating-Point No-Wait Clear Flags | 3 | | | | X87 | |
| FNINIT | Floating-Point No-Wait Initialize | 3 | | | | X87 | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FNOP | Floating-Point No Operation | 3 | | | | X87 | |
| FNSAVE | Save No-Wait x87 and MMX State | 3 | | | X87 | X87 | |
| FNSTCW | Floating-Point No-Wait Store x87 Control Word | 3 | | | | X87 | |
| FNSTENV | Floating-Point No-Wait Store x87 Environment | 3 | | | | X87 | |
| FNSTSW | Floating-Point No-Wait Store x87 Status Word | 3 | | | | X87 | |
| FPATAN | Floating-Point Partial Arctangent | 3 | | | | X87 | |
| FPREM | Floating-Point Partial Remainder | 3 | | | | X87 | |
| FPREM1 | Floating-Point Partial Remainder | 3 | | | | X87 | |
| FPTAN | Floating-Point Partial Tangent | 3 | | | | X87 | |
| FRNDINT | Floating-Point Round to Integer | 3 | | | | X87 | |
| FRSTOR | Restore x87 and MMX State | 3 | | | X87 | X87 | |
| FSAVE | Save x87 and MMX State | 3 | | | X87 | X87 | |
| FSCALE | Floating-Point Scale | 3 | | | | X87 | |
| FSIN | Floating-Point Sine | 3 | | | | X87 | |
| FSINCOS | Floating-Point Sine and Cosine | 3 | | | | X87 | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FSQRT | Floating-Point Square Root | 3 | | | | X87 | |
| FST | Floating-Point Store Stack Top | 3 | | | | X87 | |
| FSTCW | Floating-Point Store x87 Control Word | 3 | | | | X87 | |
| FSTENV | Floating-Point Store x87 Environment | 3 | | | | X87 | |
| FSTP | Floating-Point Store Stack Top and Pop | 3 | | | | X87 | |
| FSTSW | Floating-Point Store x87 Status Word | 3 | | | | X87 | |
| FSUB | Floating-Point Subtract | 3 | | | | X87 | |
| FSUBP | Floating-Point Subtract and Pop | 3 | | | | X87 | |
| FSUBR | Floating-Point Subtract Reverse | 3 | | | | X87 | |
| FSUBRP | Floating-Point Subtract Reverse and Pop | 3 | | | | X87 | |
| FTST | Floating-Point Test with Zero | 3 | | | | X87 | |
| FUCOM | Floating-Point Unordered Compare | 3 | | | | X87 | |
| FUCOMI | Floating-Point Unordered Compare and Set Flags | 3 | | | | X87 | |
| FUCOMIP | Floating-Point Unordered Compare and Set Flags and Pop | 3 | | | | X87 | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|---|-----|--|-----------------|-----------------|-----------------|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| FUCOMP | Floating-Point Unordered Compare and Pop | 3 | | | | X87 | |
| FUCOMPP | Floating-Point Unordered Compare and Pop Twice | 3 | | | | X87 | |
| FWAIT | Wait for x87 Floating-Point Exceptions | 3 | | | | X87 | |
| FXAM | Floating-Point Examine | 3 | | | | X87 | |
| FXCH | Floating-Point Exchange | 3 | | | | X87 | |
| FXRSTOR | Restore XMM, MMX, and x87 State | 3 | | FXSAVE, FXRSTOR | FXSAVE, FXRSTOR | FXSAVE, FXRSTOR | |
| FXSAVE | Save XMM, MMX, and x87 State | 3 | | FXSAVE, FXRSTOR | FXSAVE, FXRSTOR | FXSAVE, FXRSTOR | |
| FXTRACT | Floating-Point Extract Exponent and Significand | 3 | | | | X87 | |
| FYL2X | Floating-Point $y * \log_2 x$ | 3 | | | | X87 | |
| FYL2XP1 | Floating-Point $y * \log_2(x + 1)$ | 3 | | | | X87 | |
| HLT | Halt | 0 | | | | | Basic |
| IDIV | Signed Divide | 3 | Basic | | | | |
| IMUL | Signed Multiply | 3 | Basic | | | | |
| IN | Input from Port | 3 | Basic | | | | |
| INC | Increment by 1 | 3 | Basic | | | | |
| INS | Input String | 3 | Basic | | | | |
| INSB | Input String Byte | 3 | Basic | | | | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|------------------------------|-----|--|---------------|--------------|-----|-----------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| INSD | Input String Doubleword | 3 | Basic | | | | |
| INSW | Input String Word | 3 | Basic | | | | |
| INT | Interrupt to Vector | 3 | Basic | | | | |
| INT 3 | Interrupt to Debug Vector | 3 | | | | | Basic |
| INTO | Interrupt to Overflow Vector | 3 | Basic | | | | |
| INVD | Invalidate Caches | 0 | | | | | Basic |
| INVLPG | Invalidate TLB Entry | 0 | | | | | Basic |
| IRET | Interrupt Return Word | 3 | | | | | Basic |
| IRETD | Interrupt Return Doubleword | 3 | | | | | Basic |
| IRETQ | Interrupt Return Quadword | 3 | | | | | Long Mode |
| JA | Jump if Above | 3 | Basic | | | | |
| JAE | Jump if Above or Equal | 3 | Basic | | | | |
| JB | Jump if Below | 3 | Basic | | | | |
| JBE | Jump if Below or Equal | 3 | Basic | | | | |
| JC | Jump if Carry | 3 | Basic | | | | |
| JCXZ | Jump if CX Zero | 3 | Basic | | | | |
| JE | Jump if Equal | 3 | Basic | | | | |
| JECXZ | Jump if ECX Zero | 3 | Basic | | | | |
| JG | Jump if Greater | 3 | Basic | | | | |
| JGE | Jump if Greater or Equal | 3 | Basic | | | | |
| JL | Jump if Less | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|------------------------------|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| JLE | Jump if Less or Equal | 3 | Basic | | | | |
| JMP | Jump | 3 | Basic | | | | |
| JNA | Jump if Not Above | 3 | Basic | | | | |
| JNAE | Jump if Not Above or Equal | 3 | Basic | | | | |
| JNB | Jump if Not Below | 3 | Basic | | | | |
| JNBE | Jump if Not Below or Equal | 3 | Basic | | | | |
| JNC | Jump if Not Carry | 3 | Basic | | | | |
| JNE | Jump if Not Equal | 3 | Basic | | | | |
| JNG | Jump if Not Greater | 3 | Basic | | | | |
| JNGE | Jump if Not Greater Or Equal | 3 | Basic | | | | |
| JNL | Jump if Not Less | 3 | Basic | | | | |
| JNLE | Jump if Not Less or Equal | 3 | Basic | | | | |
| JNO | Jump if Not Overflow | 3 | Basic | | | | |
| JNP | Jump if Not Parity | 3 | Basic | | | | |
| JNS | Jump if Not Sign | 3 | Basic | | | | |
| JNZ | Jump if Not Zero | 3 | Basic | | | | |
| JO | Jump if Overflow | 3 | Basic | | | | |
| JP | Jump if Parity | 3 | Basic | | | | |
| JPE | Jump if Parity Even | 3 | Basic | | | | |
| JPO | Jump if Parity Odd | 3 | Basic | | | | |
| JRCXZ | Jump if RCX Zero | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| JS | Jump if Sign | 3 | Basic | | | | |
| JZ | Jump if Zero | 3 | Basic | | | | |
| LAHF | Load Status Flags into AH Register | 3 | Basic | | | | |
| LAR | Load Access Rights Byte | 3 | | | | | Basic |
| LDMXCSR | Load MXCSR Control/Status Register | 3 | | SSE | | | |
| LDS | Load DS Far Pointer | 3 | Basic | | | | |
| LEA | Load Effective Address | 3 | Basic | | | | |
| LEAVE | Delete Procedure Stack Frame | 3 | Basic | | | | |
| LES | Load ES Far Pointer | 3 | Basic | | | | |
| LFENCE | Load Fence | 3 | SSE2 | | | | |
| LFS | Load FS Far Pointer | 3 | Basic | | | | |
| LGDT | Load Global Descriptor Table Register | 0 | | | | | Basic |
| LGS | Load GS Far Pointer | 3 | Basic | | | | |
| LIDT | Load Interrupt Descriptor Table Register | 0 | | | | | Basic |
| LLDT | Load Local Descriptor Table Register | 0 | | | | | Basic |
| LMSW | Load Machine Status Word | 0 | | | | | Basic |
| LODS | Load String | 3 | Basic | | | | |
| LODSB | Load String Byte | 3 | Basic | | | | |
| LODSD | Load String Doubleword | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| LODSQ | Load String Quadword | 3 | Long Mode | | | | |
| LODSW | Load String Word | 3 | Basic | | | | |
| LOOP | Loop | 3 | Basic | | | | |
| LOOPE | Loop if Equal | 3 | Basic | | | | |
| LOOPNE | Loop if Not Equal | 3 | Basic | | | | |
| LOOPNZ | Loop if Not Zero | 3 | Basic | | | | |
| LOOPZ | Loop if Zero | 3 | Basic | | | | |
| LSL | Load Segment Limit | 3 | Basic | | | | |
| LSS | Load SS Segment Register | 3 | Basic | | | | |
| LTR | Load Task Register | 0 | | | | | Basic |
| MASKMOVDQU | Masked Move Double Quadword Unaligned | 3 | | SSE2 | | | |
| MASKMOVQ | Masked Move Quadword | 3 | | | SSE, MMX Extensions | | |
| MAXPD | Maximum Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MAXPS | Maximum Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MAXSD | Maximum Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MAXSS | Maximum Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| MFENCE | Memory Fence | 3 | SSE2 | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| MINPD | Minimum Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MINPS | Minimum Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MINSD | Minimum Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MINSS | Minimum Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| MOV | Move | 3 | Basic | | | | |
| MOV (2) | Move to/from Control Registers | 0 | | | | | Basic |
| MOV (3) | Move to/from Debug Registers | 0 | | | | | Basic |
| MOVAPD | Move Aligned Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MOVAPS | Move Aligned Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVD | Move Doubleword or Quadword | 3 | MMX, SSE2 | SSE2 | MMX | | |
| MOVDQ2Q | Move Quadword to Quadword | 3 | | SSE2 | SSE2 | | |
| MOVDQA | Move Aligned Double Quadword | 3 | | SSE2 | | | |
| MOVDQU | Move Unaligned Double Quadword | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| MOVHLP | Move Packed Single-Precision Floating-Point High to Low | 3 | | SSE | | | |
| MOVHPD | Move High Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MOVHPS | Move High Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVLHP | Move Packed Single-Precision Floating-Point Low to High | 3 | | SSE | | | |
| MOVLPD | Move Low Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MOVLPS | Move Low Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVMSKPD | Extract Packed Double-Precision Floating-Point Sign Mask | 3 | SSE2 | SSE2 | | | |
| MOVMSKPS | Extract Packed Single-Precision Floating-Point Sign Mask | 3 | SSE | SSE | | | |
| MOVNTDQ | Move Non-Temporal Double Quadword | 3 | | SSE2 | | | |
| MOVNTI | Move Non-Temporal Doubleword or Quadword | 3 | SSE2 | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|-------------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| MOVNTPD | Move Non-Temporal Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MOVNTPS | Move Non-Temporal Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVNTQ | Move Non-Temporal Quadword | 3 | | | SSE, MMX Extensions | | |
| MOVQ | Move Quadword | 3 | | SSE2 | MMX | | |
| MOVQ2DQ | Move Quadword to Quadword | 3 | | SSE2 | SSE2 | | |
| MOVS | Move String | 3 | Basic | | | | |
| MOVSB | Move String Byte | 3 | Basic | | | | |
| MOVSD | Move String Doubleword | 3 | Basic ² | | | | |
| MOVSD | Move Scalar Double-Precision Floating-Point | 3 | | SSE2 ² | | | |
| MOVSQ | Move String Quadword | 3 | Long Mode | | | | |
| MOVSS | Move Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVSW | Move String Word | 3 | Basic | | | | |
| MOVSX | Move with Sign-Extend | 3 | Basic | | | | |
| MOVFXD | Move with Sign-Extend Doubleword | 3 | Long Mode | | | | |
| MOVUPD | Move Unaligned Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| MOVUPS | Move Unaligned Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MOVZX | Move with Zero-Extend | 3 | Basic | | | | |
| MUL | Multiply Unsigned | 3 | Basic | | | | |
| MULPD | Multiply Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MULPS | Multiply Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| MULSD | Multiply Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| MULSS | Multiply Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| NEG | Two's Complement Negation | 3 | Basic | | | | |
| NOP | No Operation | 3 | Basic | | | | |
| NOT | One's Complement Negation | 3 | Basic | | | | |
| OR | Logical OR | 3 | Basic | | | | |
| ORPD | Logical Bitwise OR Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| ORPS | Logical Bitwise OR Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| OUT | Output to Port | 3 | Basic | | | | |
| OUTS | Output String | 3 | Basic | | | | |
| OUTSB | Output String Byte | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| OUTSD | Output String Doubleword | 3 | Basic | | | | |
| OUTSW | Output String Word | 3 | Basic | | | | |
| PACKSSDW | Pack with Saturation Signed Doubleword to Word | 3 | | SSE2 | MMX | | |
| PACKSSWB | Pack with Saturation Signed Word to Byte | 3 | | SSE2 | MMX | | |
| PACKUSWB | Pack with Saturation Signed Word to Unsigned Byte | 3 | | SSE2 | MMX | | |
| PADDB | Packed Add Bytes | 3 | | SSE2 | MMX | | |
| PADD | Packed Add Doublewords | 3 | | SSE2 | MMX | | |
| PADDQ | Packed Add Quadwords | 3 | | SSE2 | SSE2 | | |
| PADDSB | Packed Add Signed with Saturation Bytes | 3 | | SSE2 | MMX | | |
| PADDSW | Packed Add Signed with Saturation Words | 3 | | SSE2 | MMX | | |
| PADDUSB | Packed Add Unsigned with Saturation Bytes | 3 | | SSE2 | MMX | | |
| PADDUSW | Packed Add Unsigned with Saturation Words | 3 | | SSE2 | MMX | | |
| PADDW | Packed Add Words | 3 | | SSE2 | MMX | | |
| PAND | Packed Logical Bitwise AND | 3 | | SSE2 | MMX | | |
| PANDN | Packed Logical Bitwise AND NOT | 3 | | SSE2 | MMX | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PAVGB | Packed Average Unsigned Bytes | 3 | | SSE2 | SSE, MMX Extensions | | |
| PAVGUSB | Packed Average Unsigned Bytes | 3 | | | 3DNow! | | |
| PAVGW | Packed Average Unsigned Words | 3 | | SSE2 | SSE, MMX Extensions | | |
| PCMPEQB | Packed Compare Equal Bytes | 3 | | SSE2 | MMX | | |
| PCMPEQD | Packed Compare Equal Doublewords | 3 | | SSE2 | MMX | | |
| PCMPEQW | Packed Compare Equal Words | 3 | | SSE2 | MMX | | |
| PCMPGTB | Packed Compare Greater Than Signed Bytes | 3 | | SSE2 | MMX | | |
| PCMPGTD | Packed Compare Greater Than Signed Doublewords | 3 | | SSE2 | MMX | | |
| PCMPGTW | Packed Compare Greater Than Signed Words | 3 | | SSE2 | MMX | | |
| PEXTRW | Packed Extract Word | 3 | | SSE2 | SSE, MMX Extensions | | |
| PF2ID | Packed Floating-Point to Integer Doubleword Conversion | 3 | | | 3DNow! | | |
| PF2IW | Packed Floating-Point to Integer Word Conversion | 3 | | | 3DNow! Extensions | | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.
2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|----------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PFACC | Packed Floating-Point Accumulate | 3 | | | 3DNow! | | |
| PFADD | Packed Floating-Point Add | 3 | | | 3DNow! | | |
| PFCMPEQ | Packed Floating-Point Compare Equal | 3 | | | 3DNow! | | |
| PFCMPGE | Packed Floating-Point Compare Greater or Equal | 3 | | | 3DNow! | | |
| PFCMPGT | Packed Floating-Point Compare Greater Than | 3 | | | 3DNow! | | |
| PFMAX | Packed Floating-Point Maximum | 3 | | | 3DNow! | | |
| PFCMIN | Packed Floating-Point Minimum | 3 | | | 3DNow! | | |
| PFMUL | Packed Floating-Point Multiply | 3 | | | 3DNow! | | |
| PFNACC | Packed Floating-Point Negative Accumulate | 3 | | | 3DNow! Extensions | | |
| PFPNACC | Packed Floating-Point Positive-Negative Accumulate | 3 | | | 3DNow! Extensions | | |
| PFRCP | Packed Floating-Point Reciprocal Approximation | 3 | | | 3DNow! | | |
| PFRCPIT1 | Packed Floating-Point Reciprocal, Iteration 1 | 3 | | | 3DNow! | | |
| PFRCPIT2 | Packed Floating-Point Reciprocal or Reciprocal Square Root, Iteration 2 | 3 | | | 3DNow! | | |
| <ol style="list-style-type: none"> 1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. 2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PFRSQIT1 | Packed Floating-Point Reciprocal Square Root, Iteration 1 | 3 | | | 3DNow! | | |
| PFRSQRT | Packed Floating-Point Reciprocal Square Root Approximation | 3 | | | 3DNow! | | |
| PFSUB | Packed Floating-Point Subtract | 3 | | | 3DNow! | | |
| PFSUBR | Packed Floating-Point Subtract Reverse | 3 | | | 3DNow! | | |
| PI2FD | Packed Integer to Floating-Point Doubleword Conversion | 3 | | | 3DNow! | | |
| PI2FW | Packed Integer To Floating-Point Word Conversion | 3 | | | 3DNow! Extensions | | |
| PINSRW | Packed Insert Word | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMADDWD | Packed Multiply Words and Add Doublewords | 3 | | SSE2 | MMX | | |
| PMAXSW | Packed Maximum Signed Words | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMAXUB | Packed Maximum Unsigned Bytes | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMINSW | Packed Minimum Signed Words | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMINUB | Packed Minimum Unsigned Bytes | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMOVMASKB | Packed Move Mask Byte | 3 | | SSE2 | SSE, MMX Extensions | | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.

2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PMULHRW | Packed Multiply High Rounded Word | 3 | | | 3DNow! | | |
| PMULHUW | Packed Multiply High Unsigned Word | 3 | | SSE2 | SSE, MMX Extensions | | |
| PMULHW | Packed Multiply High Signed Word | 3 | | SSE2 | MMX | | |
| PMULLW | Packed Multiply Low Signed Word | 3 | | SSE2 | MMX | | |
| PMULUDQ | Packed Multiply Unsigned Doubleword and Store Quadword | 3 | | SSE2 | SSE2 | | |
| POP | Pop Stack | 3 | Basic | | | | |
| POPA | Pop All to GPR Words | 3 | Basic | | | | |
| POPAD | Pop All to GPR Doublewords | 3 | Basic | | | | |
| POPF | Pop to FLAGS Word | 3 | Basic | | | | |
| POPFQ | Pop to EFLAGS Doubleword | 3 | Basic | | | | |
| POPFQ | Pop to RFLAGS Quadword | 3 | Long Mode | | | | |
| POR | Packed Logical Bitwise OR | 3 | | SSE2 | MMX | | |
| PREFETCH | Prefetch L1 Data-Cache Line | 3 | 3DNow! | | | | |
| PREFETCH ^{level} | Prefetch Data to Cache Level ^{level} | 3 | SSE, MMX Extensions | | | | |
| PREFETCHW | Prefetch L1 Data-Cache Line for Write | 3 | 3DNow! | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|---------------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PSADBW | Packed Sum of Absolute Differences of Bytes into a Word | 3 | | SSE2 | SSE, MMX Extensions | | |
| PSHUFD | Packed Shuffle Doublewords | 3 | | SSE2 | | | |
| PSHUFHW | Packed Shuffle High Words | 3 | | SSE2 | | | |
| PSHUFLW | Packed Shuffle Low Words | 3 | | SSE2 | | | |
| PSHUFW | Packed Shuffle Words | 3 | | | SSE, MMX Extensions | | |
| PSLLD | Packed Shift Left Logical Doublewords | 3 | | SSE2 | MMX | | |
| PSLLDQ | Packed Shift Left Logical Double Quadword | 3 | | SSE2 | | | |
| PSLLQ | Packed Shift Left Logical Quadwords | 3 | | SSE2 | MMX | | |
| PSLLW | Packed Shift Left Logical Words | 3 | | SSE2 | MMX | | |
| PSRAD | Packed Shift Right Arithmetic Doublewords | 3 | | SSE2 | MMX | | |
| PSRAW | Packed Shift Right Arithmetic Words | 3 | | SSE2 | MMX | | |
| PSRLD | Packed Shift Right Logical Doublewords | 3 | | SSE2 | MMX | | |
| PSRLDQ | Packed Shift Right Logical Double Quadword | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|---|------------------|----------------------|-----|--------|
| Mnemonic | Description | CPL | General- Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PSRLQ | Packed Shift Right Logical Quadwords | 3 | | SSE2 | MMX | | |
| PSRLW | Packed Shift Right Logical Words | 3 | | SSE2 | MMX | | |
| PSUBB | Packed Subtract Bytes | 3 | | SSE2 | MMX | | |
| PSUBD | Packed Subtract Doublewords | 3 | | SSE2 | MMX | | |
| PSUBQ | Packed Subtract Quadword | 3 | | SSE2 | SSE2 | | |
| PSUBSB | Packed Subtract Signed With Saturation Bytes | 3 | | SSE2 | MMX | | |
| PSUBSW | Packed Subtract Signed with Saturation Words | 3 | | SSE2 | MMX | | |
| PSUBUSB | Packed Subtract Unsigned and Saturate Bytes | 3 | | SSE2 | MMX | | |
| PSUBUSW | Packed Subtract Unsigned and Saturate Words | 3 | | SSE2 | MMX | | |
| PSUBW | Packed Subtract Words | 3 | | SSE2 | MMX | | |
| PSWAPD | Packed Swap Doubleword | 3 | | | 3DNow! Extensions | | |
| PUNPCKHBW | Unpack and Interleave High Bytes | 3 | | SSE2 | MMX | | |
| PUNPCKHDQ | Unpack and Interleave High Doublewords | 3 | | SSE2 | MMX | | |
| PUNPCKHQDQ | Unpack and Interleave High Quadwords | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| PUNPCKHWD | Unpack and Interleave High Words | 3 | | SSE2 | MMX | | |
| PUNPCKLBW | Unpack and Interleave Low Bytes | 3 | | SSE2 | MMX | | |
| PUNPCKLDQ | Unpack and Interleave Low Doublewords | 3 | | SSE2 | MMX | | |
| PUNPCKLQDQ | Unpack and Interleave Low Quadwords | 3 | | SSE2 | | | |
| PUNPCKLWD | Unpack and Interleave Low Words | 3 | | SSE2 | MMX | | |
| PUSH | Push onto Stack | 3 | Basic | | | | |
| PUSHA | Push All GPR Words onto Stack | 3 | Basic | | | | |
| PUSHAD | Push All GPR Doublewords onto Stack | 3 | Basic | | | | |
| PUSHF | Push EFLAGS Word onto Stack | 3 | Basic | | | | |
| PUSHFD | Push EFLAGS Doubleword onto Stack | 3 | Basic | | | | |
| PUSHFQ | Push RFLAGS Quadword onto Stack | 3 | Long Mode | | | | |
| PXOR | Packed Logical Bitwise Exclusive OR | 3 | | SSE2 | MMX | | |
| RCL | Rotate Through Carry Left | 3 | Basic | | | | |
| RCPPS | Reciprocal Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|---|-----|--|---------------|--------------|-----|--------------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| RCPSS | Reciprocal Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| RCR | Rotate Through Carry Right | 3 | Basic | | | | |
| RDMSR | Read Model-Specific Register | 0 | | | | | RDMSR, WRMSR |
| RDPMC | Read Performance-Monitoring Counter | 3 | | | | | Basic |
| RDTSC | Read Time-Stamp Counter | 3 | | | | | Basic |
| RET | Return from Call | 3 | Basic | | | | |
| ROL | Rotate Left | 3 | Basic | | | | |
| ROR | Rotate Right | 3 | Basic | | | | |
| RSM | Resume from System Management Mode | 0 | | | | | Basic |
| RSQRTPS | Reciprocal Square Root Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| RSQRTSS | Reciprocal Square Root Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| SAHF | Store AH into Flags | 3 | Basic | | | | |
| SAL | Shift Arithmetic Left | 3 | Basic | | | | |
| SAR | Shift Arithmetic Right | 3 | Basic | | | | |
| SBB | Subtract with Borrow | 3 | Basic | | | | |
| SCAS | Scan String | 3 | Basic | | | | |
| SCASB | Scan String as Bytes | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--------------------------------|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| SCASD | Scan String as Doubleword | 3 | Basic | | | | |
| SCASQ | Scan String as Quadword | 3 | Long Mode | | | | |
| SCASW | Scan String as Words | 3 | Basic | | | | |
| SETA | Set Byte if Above | 3 | Basic | | | | |
| SETAE | Set Byte if Above or Equal | 3 | Basic | | | | |
| SETB | Set Byte if Below | 3 | Basic | | | | |
| SETBE | Set Byte if Below or Equal | 3 | Basic | | | | |
| SETC | Set Byte if Carry | 3 | Basic | | | | |
| SETE | Set Byte if Equal | 3 | Basic | | | | |
| SETG | Set Byte if Greater | 3 | Basic | | | | |
| SETGE | Set Byte if Greater or Equal | 3 | Basic | | | | |
| SETL | Set Byte if Less | 3 | Basic | | | | |
| SETLE | Set Byte if Less or Equal | 3 | Basic | | | | |
| SETNA | Set Byte if Not Above | 3 | Basic | | | | |
| SETNAE | Set Byte if Not Above or Equal | 3 | Basic | | | | |
| SETNB | Set Byte if Not Below | 3 | Basic | | | | |
| SETNBE | Set Byte if Not Below or Equal | 3 | Basic | | | | |
| SETNC | Set Byte if No Carry | 3 | Basic | | | | |
| SETNE | Set Byte if Not Equal | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|-------------|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| SETNG | Set Byte if Not Greater | 3 | Basic | | | | |
| SETNGE | Set Byte if Not Greater or Equal | 3 | Basic | | | | |
| SETNL | Set Byte if Not Less | 3 | Basic | | | | |
| SETNLE | Set Byte if Not Less or Equal | 3 | Basic | | | | |
| SETNO | Set Byte if Not Overflow | 3 | Basic | | | | |
| SETNP | Set Byte if Not Parity | 3 | Basic | | | | |
| SETNS | Set Byte if Not Sign | 3 | Basic | | | | |
| SETNZ | Set Byte if Not Zero | 3 | Basic | | | | |
| SETO | Set Byte if Overflow | 3 | Basic | | | | |
| SETP | Set Byte if Parity | 3 | Basic | | | | |
| SETPE | Set Byte if Parity Even | 3 | Basic | | | | |
| SETPO | Set Byte if Parity Odd | 3 | Basic | | | | |
| SETS | Set Byte if Sign | 3 | Basic | | | | |
| SETZ | Set Byte if Zero | 3 | Basic | | | | |
| SFENCE | Store Fence | 3 | SSE, MMX Extensions | | | | |
| SGDT | Store Global Descriptor Table Register | 3 | | | | | Basic |
| SHL | Shift Left | 3 | Basic | | | | |
| SHLD | Shift Left Double | 3 | Basic | | | | |
| SHR | Shift Right | 3 | Basic | | | | |
| SHRD | Shift Right Double | 3 | Basic | | | | |

1. Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs.

2. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands.

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| SHUFPD | Shuffle Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| SHUFPS | Shuffle Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| SIDT | Store Interrupt Descriptor Table Register | 3 | | | | | Basic |
| SLDT | Store Local Descriptor Table Register | 3 | | | | | Basic |
| SMSW | Store Machine Status Word | 3 | | | | | Basic |
| SQRTPD | Square Root Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| SQRTPS | Square Root Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| SQRTSD | Square Root Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| SQRTSS | Square Root Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| STC | Set Carry Flag | 3 | Basic | | | | |
| STD | Set Direction Flag | 3 | Basic | | | | |
| STI | Set Interrupt Flag | 3 | | | | | Basic |
| STMXCSR | Store MXCSR Control/Status Register | 3 | | SSE | | | |
| STOS | Store String | 3 | Basic | | | | |
| STOSB | Store String Bytes | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|-------------------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| STOSD | Store String Doublewords | 3 | Basic | | | | |
| STOSQ | Store String Quadwords | 3 | Long Mode | | | | |
| STOSW | Store String Words | 3 | Basic | | | | |
| STR | Store Task Register | 3 | | | | | Basic |
| SUB | Subtract | 3 | Basic | | | | |
| SUBPD | Subtract Packed Double-Precision Floating-Point | 3 | | SSE2 | | | |
| SUBPS | Subtract Packed Single-Precision Floating-Point | 3 | | SSE | | | |
| SUBSD | Subtract Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| SUBSS | Subtract Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| SWAPGS | Swap GS Register with KernelGSbase MSR | 0 | | | | | Long Mode |
| SYSCALL | Fast System Call | 3 | | | | | SYSCALL, SYSRET |
| SYSENTER | System Call | 3 | | | | | SYSENTER, SYSEXIT |
| SYSEXIT | System Return | 0 | | | | | SYSENTER, SYSEXIT |
| SYSRET | Fast System Return | 0 | | | | | SYSCALL, SYSRET |
| TEST | Test Bits | 3 | Basic | | | | |
| UCOMISD | Unordered Compare Scalar Double-Precision Floating-Point | 3 | | SSE2 | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|--|---------------|--------------|-----|--------------|
| Mnemonic | Description | CPL | General-Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| UCOMISS | Unordered Compare Scalar Single-Precision Floating-Point | 3 | | SSE | | | |
| UD2 | Undefined Operation | 3 | | | | | Basic |
| UNPCKHPD | Unpack High Double-Precision Floating-Point | 3 | | SSE2 | | | |
| UNPCKHPS | Unpack High Single-Precision Floating-Point | 3 | | SSE | | | |
| UNPCKLPD | Unpack Low Double-Precision Floating-Point | 3 | | SSE2 | | | |
| UNPCKLPS | Unpack Low Single-Precision Floating-Point | 3 | | SSE | | | |
| VERR | Verify Segment for Reads | 3 | | | | | Basic |
| VERW | Verify Segment for Writes | 3 | | | | | Basic |
| WAIT | Wait for x87 Floating-Point Exceptions | 3 | | | | X87 | |
| WBINVD | Writeback and Invalidate Caches | 0 | | | | | Basic |
| WRMSR | Write to Model-Specific Register | 0 | | | | | RDMSR, WRMSR |
| XADD | Exchange and Add | 3 | Basic | | | | |
| XCHG | Exchange | 3 | Basic | | | | |
| XLAT | Translate Table Index | 3 | Basic | | | | |
| XLATB | Translate Table Index (No Operands) | 3 | Basic | | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Table D-1. Instruction Subsets and CPUID Feature Sets (continued)

| Instruction | | | Instruction Subset and CPUID Feature Set(s) ¹ | | | | |
|---|--|-----|---|------------------|-----------------|-----|--------|
| Mnemonic | Description | CPL | General- Purpose | 128-Bit Media | 64-Bit Media | x87 | System |
| XOR | Exclusive OR | 3 | Basic | | | | |
| XORPD | Logical Bitwise Exclusive OR Packed Double- Precision Floating-Point | 3 | | SSE2 | | | |
| XORPS | Logical Bitwise Exclusive OR Packed Single- Precision Floating-Point | 3 | | SSE | | | |
| <ol style="list-style-type: none"> Columns indicate the instruction subsets. Entries indicate the CPUID feature set(s) to which the instruction belongs. Mnemonic is used for two different instructions. Assemblers can distinguish them by the number and type of operands. | | | | | | | |

Appendix E Instruction Effects on RFLAGS

The flags in the RFLAGS register are described in “Flags Register” in Volume 1 and “RFLAGS Register” in Volume 2. Table E-1 summarizes the effect that instructions have on these flags. The table includes all instructions that affect the flags. Instructions not shown have no effect on RFLAGS.

The following codes are used within the table:

- 0—The flag is or can be cleared to 0.
- 1—The flag is or can be set to 1.
- Load—The flag is loaded with value from AH register.
- Mod—The flag is or can be modified.
- Pop—The flag is loaded with value popped off of the stack.
- Tst—The flag is tested.
- U—The effect on the flag is undefined.
- Gray shaded cells indicate that the flag is not affected by the instruction.

Table E-1. Instruction Effects on RFLAGS

| Instruction Mnemonic | RFLAGS Mnemonic and Bit Number | | | | | | | | | | | | | | | |
|----------------------|--------------------------------|--------|--------|-------|-------|-------|-------|-------|-------|------|------|------|------|------------|------|------------|
| | ID 21 | VIP 20 | VIF 19 | AC 18 | VM 17 | RF 16 | NT 14 | OF 11 | DF 10 | IF 9 | TF 8 | SF 7 | ZF 6 | AF 4 | PF 2 | CF 0 |
| AAA AAS | | | | | | | | U | | | | U | U | Tst Mod | U | Mod |
| AAD AAM | | | | | | | | U | | | | Mod | Mod | U | Mod | U |
| ADC | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Tst Mod |
| ADD | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| AND | | | | | | | | 0 | | | | Mod | Mod | U | Mod | 0 |
| ARPL | | | | | | | | | | | | | Mod | | | |
| BSF BSR | | | | | | | | U | | | | U | Mod | U | U | U |

Table E-1. Instruction Effects on RFLAGS (continued)

| Instruction Mnemonic | RFLAGS Mnemonic and Bit Number | | | | | | | | | | | | | | | |
|--------------------------------------|--------------------------------|--------|--------|-------|-------|-------|-------|-------|-------|------|------|------|------|------------|------|------------|
| | ID 21 | VIP 20 | VIF 19 | AC 18 | VM 17 | RF 16 | NT 14 | OF 11 | DF 10 | IF 9 | TF 8 | SF 7 | ZF 6 | AF 4 | PF 2 | CF 0 |
| BT BTC BTR BTS | | | | | | | | U | | | | U | U | U | U | Mod |
| CLC | | | | | | | | | | | | | | | | 0 |
| CLD | | | | | | | | | 0 | | | | | | | |
| CLI | | | | | | | | | | 0 | | | | | | |
| CMC | | | | | | | | | | | | | | | | Mod |
| CMOV _{cc} | | | | | | | | Tst | | | | Tst | Tst | | Tst | Tst |
| CMP | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| CMPS _x | | | | | | | | Mod | Tst | | | Mod | Mod | Mod | Mod | Mod |
| CMPXCHG | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| CMPXCHG 8B | | | | | | | | | | | | | | Mod | | |
| COMISD COMISS | | | | | | | | 0 | | | | 0 | Mod | 0 | Mod | Mod |
| DAA DAS | | | | | | | | U | | | | Mod | Mod | Tst Mod | Mod | Tst Mod |
| DEC | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | |
| DIV | | | | | | | | U | | | | U | U | U | U | U |
| FCMOV _{cc} | | | | | | | | | | | | | Tst | | Tst | Tst |
| FCOMI FCOMIP FUCOMI FUCOMIP | | | | | | | | | | | | | Mod | | Mod | Mod |
| IDIV | | | | | | | | U | | | | U | U | U | U | U |
| IMUL | | | | | | | | Mod | | | | U | U | U | U | Mod |
| INC | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | |
| INS _x | | | | | | | | | Tst | | | | | | | |

Table E-1. Instruction Effects on RFLAGS (continued)

| Instruction Mnemonic | RFLAGS Mnemonic and Bit Number | | | | | | | | | | | | | | | |
|--|--------------------------------|--------|--------|-------|-------|-------|-------|-------|-------|------|------|------|------|------|------|------------|
| | ID 21 | VIP 20 | VIF 19 | AC 18 | VM 17 | RF 16 | NT 14 | OF 11 | DF 10 | IF 9 | TF 8 | SF 7 | ZF 6 | AF 4 | PF 2 | CF 0 |
| INT INT 3 | | | | 0 | 0 | 0 | 0 | | | 0 | 0 | | | | | |
| INTO | | | | | | | 0 | Tst | | | 0 | | | | | |
| IRET _x | | | | | | | Tst | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop |
| Jcc | | | | | | | | Tst | | | | Tst | Tst | | Tst | Tst |
| LAR | | | | | | | | | | | | | Mod | | | |
| LODS _x | | | | | | | | | Tst | | | | | | | |
| LOOPE LOOPNE | | | | | | | | | | | | | Tst | | | |
| LSL | | | | | | | | | | | | | Mod | | | |
| MOV(CR _n) MOV(DR _n) | | | | | | | | U | | | | U | U | U | U | U |
| MOVS _x | | | | | | | | | Tst | | | | | | | |
| MUL | | | | | | | | Mod | | | | U | U | U | U | Mod |
| NEG | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| OR | | | | | | | | 0 | | | | Mod | Mod | U | Mod | 0 |
| OUTS _x | | | | | | | | | Tst | | | | | | | |
| POPF _x | | | | | | | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop |
| RCL 1 | | | | | | | | Mod | | | | | | | | Tst Mod |
| RCL <i>count</i> | | | | | | | | U | | | | | | | | Tst Mod |
| RCR 1 | | | | | | | | Mod | | | | | | | | Tst Mod |
| RCR <i>count</i> | | | | | | | | U | | | | | | | | Tst Mod |
| ROL 1 | | | | | | | | Mod | | | | | | | | Mod |
| ROL <i>count</i> | | | | | | | | U | | | | | | | | Mod |

Table E-1. Instruction Effects on RFLAGS (continued)

| Instruction Mnemonic | RFLAGS Mnemonic and Bit Number | | | | | | | | | | | | | | | |
|----------------------|--------------------------------|--------|--------|-------|-------|-------|-------|-------|-------|------|------|------|------|------|------|---------|
| | ID 21 | VIP 20 | VIF 19 | AC 18 | VM 17 | RF 16 | NT 14 | OF 11 | DF 10 | IF 9 | TF 8 | SF 7 | ZF 6 | AF 4 | PF 2 | CF 0 |
| ROR 1 | | | | | | | | Mod | | | | | | | | Mod |
| ROR <i>count</i> | | | | | | | | U | | | | | | | | Mod |
| RSM | | | | | | Mod | Mod | Mod | Mod | Mod | Mod | Mod | Mod | Mod | Mod | Mod |
| SAHF | | | | | | | | | | | | Load | Load | Load | Load | Load |
| SAL 1 | | | | | | | | Mod | | | | Mod | Mod | U | Mod | Mod |
| SAL <i>count</i> | | | | | | | | U | | | | Mod | Mod | U | Mod | Mod |
| SAR 1 | | | | | | | | Mod | | | | Mod | Mod | U | Mod | Mod |
| SAR <i>count</i> | | | | | | | | U | | | | Mod | Mod | U | Mod | Mod |
| SBB | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Tst Mod |
| SCASx | | | | | | | | Mod | Tst | | | Mod | Mod | Mod | Mod | Mod |
| SETcc | | | | | | | | Tst | | | | Tst | Tst | | Tst | Tst |
| SHLD SHRD | | | | | | | | U | | | | Mod | Mod | U | Mod | Mod |
| STC | | | | | | | | | | | | | | | | 1 |
| STD | | | | | | | | | 1 | | | | | | | |
| STI | | | | | | | | | | 1 | | | | | | |
| STOSx | | | | | | | | | Tst | | | | | | | |
| SUB | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| SYSCALL | | | | | 0 | | | | | 0 | | | | | | |
| SYSENTER | | | | | 0 | 0 | | | | 0 | | | | | | |
| SYSRET | | | | | | | | | | 1 | | | | | | |
| TEST | | | | | | | | 0 | | | | Mod | Mod | U | Mod | 0 |
| UCOMISD UCOMISS | | | | | | | | 0 | | | | 0 | Mod | 0 | Mod | Mod |

Table E-1. Instruction Effects on RFLAGS (continued)

| Instruction Mnemonic | RFLAGS Mnemonic and Bit Number | | | | | | | | | | | | | | | |
|----------------------|--------------------------------|-----------|-----------|----------|----------|----------|----------|----------|----------|---------|---------|---------|---------|---------|---------|---------|
| | ID 21 | VIP 20 | VIF 19 | AC 18 | VM 17 | RF 16 | NT 14 | OF 11 | DF 10 | IF 9 | TF 8 | SF 7 | ZF 6 | AF 4 | PF 2 | CF 0 |
| VERR VERW | | | | | | | | | | | | | Mod | | | |
| XADD | | | | | | | | Mod | | | | Mod | Mod | Mod | Mod | Mod |
| XOR | | | | | | | | 0 | | | | Mod | Mod | U | Mod | 0 |

Index

Numerics

| | |
|------------------|----|
| 16-bit mode..... | xv |
| 32-bit mode..... | xv |
| 64-bit mode..... | xv |

A

| | |
|---------------------------|--------------------|
| AAA | 58 |
| AAD | 59 |
| AAM..... | 61 |
| AAS | 62 |
| ADC..... | 63 |
| ADD | 66 |
| address size prefix | 6, 24 |
| addressing | |
| byte registers..... | 17 |
| effective address..... | 412, 414, 415, 417 |
| PC-relative | 23 |
| RIP-relative | xx, 23 |
| AND | 68 |
| ARPL | 300 |

B

| | |
|----------------------------------|----------|
| base field | 416, 417 |
| biased exponent..... | xv |
| BOUND | 71 |
| BSF | 73 |
| BSR | 75 |
| BSWAP | 76 |
| BT | 77 |
| BTC | 79 |
| BTR | 81 |
| BTS..... | 83 |
| byte order of instructions | 1 |
| byte register addressing..... | 17 |

C

| | |
|--------------------------------------|----------|
| cache configuration information..... | 125 |
| CALL..... | 15, 85 |
| CBW | 92 |
| CDQ..... | 93 |
| CDQE..... | 94 |
| CLC | 95 |
| CLD | 96 |
| CLFLUSH..... | 97, 397 |
| CLI | 302 |
| CLTS..... | 304 |
| CMC | 99 |
| CMOVcc..... | 100, 393 |
| CMP | 104 |
| CMPSx | 107 |

| | |
|------------------------------|----------|
| CMPXCHG..... | 110 |
| CMPXCHG8B | 15, 112 |
| commit..... | xv |
| compatibility mode..... | xv |
| condition codes | |
| rFLAGS | 393, 409 |
| count..... | 420 |
| CPUID | 114 |
| extended functions | 114 |
| feature sets | 461 |
| standard functions | 114 |
| CPUID instruction | |
| cache information | 125 |
| long-mode address sizes..... | 128 |
| testing for..... | 114 |
| CQO | 130 |
| CWD | 131 |
| CWDE..... | 132 |

D

| | |
|-------------------------|--------------|
| DAA | 133 |
| DAS..... | 134 |
| data types | |
| 128-bit media | 34 |
| 64-bit media | 36 |
| general-purpose..... | 30 |
| x87 | 38 |
| DEC | 17, 135, 454 |
| direct referencing..... | xvi |
| displacements..... | xvi, 22, 420 |
| DIV | 137 |
| double quadword..... | xvi |
| doubleword | xvi |

E

| | |
|------------------------------|--------------------|
| eAX–eSP register | xxii |
| effective address | 412, 414, 415, 417 |
| effective address size..... | xvii |
| effective operand size | xvii |
| eFLAGS register | xxii |
| eIP register | xxiii |
| element | xvii |
| endian order | xxv, 1 |
| ENTER | 139 |
| exceptions..... | xvii, 39 |
| exponent | xv |

F

| | |
|--------------|------|
| FCMOVcc..... | 409 |
| flush..... | xvii |

G

general-purpose registers 28

H

HLT 305

I

IDIV 141

IGN xvii

immediate operands 22, 420

IMUL 143

IN 146

INC 17, 147, 454

index field 417

indirect xviii

instructions

128-bit media 463

3DNow!™ 461

64-bit media 463

byte order 1

effects on rFLAGS 503

formats 1

general-purpose 57, 463

invalid in 64-bit mode 450

invalid in long mode 451

MMX™ 461

opcodes 19, 379

origins 459

reassigned in 64-bit mode 451

SSE 462

SSE-2 462

subsets 25, 459

system 299, 463

x87 461, 463

INSx 149

INT 152

INT 3 306

interrupt vectors 39

INTO 159

INVD 312

INVLPG 313, 397

J

Jcc 15, 162, 393

JCXZ 15, 166

JECXZ 15

JMP 15, 167

JRCXZ 15

L

LAHF 173

LAR 320

LDS 174

LEA 176

LEAVE 178

legacy mode xviii

legacy x86 xviii

LES 180

LFENCE 182

LFS 183

LGDT 323

LGS 185

LIDT 325

LLDT 327

LMSW 329

LOCK prefix 10

LODSx 187

long mode xviii

long-mode address sizes 128

LOOPcc 15

LOOPx 189

LSB xviii

lsb xviii

LSL 330

LSS 191

LTR 333

M

mask xix

MBZ xix

MFENCE 193

mod field 394, 412

modes 457

16-bit xv

32-bit xv

64-bit xv, 457

compatibility xv, 457

legacy xviii

long xviii, 457

protected xx

real xx

virtual-8086 xxii

ModRM byte 19, 20, 24, 394, 400, 409

moffset xix

MOV 15, 194

MOVD 198

MOVMSKPD 201

MOVMSKPS 203

MOVNTI 205

MOVSB 209

MOVSD 206

MOVSS 210

MOVZB 211

MSB xix

msb xix

MSR xxiii

| | | | |
|------------------------------|------------------|----------------------------------|-------------------------|
| MUL | 213 | prefixes | |
| N | | address size | 6, 24 |
| NEG | 215 | LOCK | 10 |
| NOP | 217, 454 | operand size | 5 |
| NOT | 218 | repeat | 10 |
| notation | 41, 379 | REX | 14, 24 |
| O | | segment | 9 |
| octword | xix | processor feature identification | |
| offset | xix, 22 | (rFLAGS.ID) | 114 |
| opcodes | 19 | processor name | 124 |
| 3DNow!™ | 397 | processor signature | 121 |
| group 1 | 395 | processor vendor | 115, 121 |
| group 10 | 396 | processor version | 116 |
| group 11 | 396 | protected mode | xx |
| group 12 | 396 | PUSH | 238 |
| group 13 | 396 | PUSHAx | 240 |
| group 14 | 396 | PUSHFcx | 241 |
| group 15 | 396 | Q | |
| group 16 | 397 | quadword | xx |
| group 1a | 395 | R | |
| group 2 | 395 | r/m field | 394, 397 |
| group 3 | 395 | r8–r15 | xxiii |
| group 4 | 395 | rAX–rSP | xxiii |
| group 5 | 395 | RAZ | xx |
| group 6 | 395 | RCL | 243 |
| group 7 | 396 | RCR | 245 |
| group 8 | 396 | RDMSR | 339 |
| group 9 | 396 | RDPMSR | 340 |
| group P | 397 | RDTSC | 341 |
| groups | 394 | real address mode. See real mode | |
| ModRM byte | 394 | real mode | xx |
| one-byte | 381 | reg field | 394, 400, 411, 412, 413 |
| two-byte | 384 | registers | |
| x87 | 400 | eAX–eSP | xxii |
| operands | | eFLAGS | xxii |
| encodings | 409 | eIP | xxiii |
| immediate | 22, 420 | encodings | 17 |
| size | 5, 419, 420, 452 | general-purpose | 28 |
| OR | 220 | MMX | 36 |
| OUT | 223 | r8–r15 | xxiii |
| OUTSx | 224 | rAX–rSP | xxiii |
| overflow | xix | rFLAGS | xxiv, 393, 409, 503 |
| P | | rIP | xxiv |
| packed | xx | segment | 30 |
| PC-relative addressing | 23 | system | 31 |
| POP | 226 | x87 | 38 |
| POPAx | 229 | XMM | 33 |
| POPcx | 231 | relative | xx |
| PREFETCHlevel | 236 | REPx prefixes | 10 |
| PREFETCHx | 234 | RET | 15, 247 |
| | | REX prefixes | 14, 24, 409 |

REX.B bit 17, 45, 412, 416
 REX.R bit 16, 411
 REX.W bit 16
 REX.X bit 16
 rFLAGS conditions codes..... 393, 409
 rFLAGS register..... xxiv, 503
 rIP register xxiv
 RIP-relative addressing..... xx, 23
 ROL 251
 ROR..... 253
 rotate count 420
 RSM 342

S

SAHF 255
 SAL 256
 SAR 258
 SBB 261
 scale field 417
 SCASx 264
 segment prefixes..... 9, 455
 segment registers..... 30
 set xx
 SETcc 267, 393
 SFENCE..... 269, 397
 SGDT 343
 shift count 420
 SHL 270
 SHLD 272
 SHR 275
 SHRD 277
 SIB byte 19, 21, 24, 415
 SSE xxi
 SSE-2 xxi
 STC 280
 STD 281
 sticky bits xxi
 STOSx 282
 SUB 284
 SWAPGS 397
 syntax..... 40
 SYSCALL..... 357
 SYSENTER..... 362
 SYSEXIT..... 364
 SYSRET 366
 system data structures 32

T

TEST 287
 TSS xxi

U

underflow xxi

V

vector xxi
 virtual-8086 mode xxii

X

XADD 289
 XCHG 291
 XLATx 293
 XOR 295

Z

zero-extension..... 420